UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

TRABAJO DE FINAL DE GRADO

Grado en Ingeniería informática (GEI) - Ingeniería del Software

Análisis e implementación de una gestión dinámica de la granularidad de las tareas en un sistema distribuido

UNIVERSIDAD POLITÉCNICA DE CATALUÑA (UPC) – BarcelonaTech

FACULTAD DE INFORMÁTICA DE BARCELONA (FIB)

Autor: Adrián Espejo Saldaña

Directora: Yolanda Becerra - Departamento de Arquitectura de Computadores

Convocatoria: Primavera 2019

Fecha de defensa: 01/07/2019

Con la colaboración del Barcelona Supercomputing Center (BSC)

Barcelona
Supercomputing
Center
Centro Nacional de Supercomputación

Resumen

Hecuba es una interfaz y un conjunto de herramientas escrito en Python y C++, desarrollada por el equipo Data-driven Scientific Computing del Barcelona Supercomputing Center [1]. El objetivo es facilitar a los programadores una interacción fácil y eficiente con tecnologías no relacionales, en este caso la base de datos distribuida Cassandra [2].

Hecuba también se puede integrar con PyCOMPSs, un framework que facilita el desarrollo y ejecución de aplicaciones en paralelo en infraestructuras distribuidas [3]. Esta integración se beneficia de la distribución de los datos que ofrece Hecuba mediante Cassandra, para su posterior computación distribuida con PyCOMPSs. Para facilitar este procesamiento distribuido, otro uso de Hecuba y el más importante para este proyecto es el particionador de datos. Este divide un objeto (que puede asociarse, por ejemplo, a una tabla de la base de datos) en otros más pequeños que contienen subconjuntos de los datos. **El término granularidad en este proyecto se refiere al tamaño de los subobjetos creados**. Después de dividir los datos se crearán tantas tareas como particiones, que se ejecutarán de forma distribuida gracias a PyCOMPSs.

Cuando se distribuye una aplicación es esencial elegir la granularidad de las tareas cuidadosamente. En primer lugar, si la granularidad es muy pequeña, el overhead de creación de tareas y de sincronización puede ser muy grande. En segundo lugar, si la granularidad es demasiado grande, es posible que no se paralelice lo suficiente. Estos son conocimientos previos que con Hecuba los programadores no deberían necesitar.

Este proyecto aspira a implementar un gestor de la granularidad dinámico integrado con Hecuba y PyCOMPSs, con el que los programadores no tengan que intervenir en la distribución de los datos, y aun así consiguiendo que las aplicaciones sean lo más eficientes posible.

Resum

Hecuba és una interfície i un conjunt d'eines escrit en Python i C++, desenvolupada per l'equip Data-driven Scientific Computing del Barcelona Supercomputing Center. L'objectiu és facilitar als programadors una interacció fàcil i eficient amb tecnologies no relacionals, en aquest cas la base de dades distribuïda Cassandra.

Hecuba també es pot integrar amb PyCOMPSs, un framework que facilita el desenvolupament i execució d'aplicacions en paral·lel en infraestructures distribuïdes. Aquesta integració es beneficia de la distribució de les dades que ofereix Hecuba mitjançant Cassandra, per a la seva posterior computació distribuïda amb PyCOMPSs. Per facilitar aquest processament distribuït, un altre ús d'Hecuba i el més important per a aquest projecte és el particionador de dades. Aquest divideix un objecte (que pot associar-se, per exemple, a una taula de la base de dades) en altres més petits que contenen subconjunts de les dades. **El terme granularitat en aquest projecte es refereix a la mida dels subobjectes creats**. Després de dividir les dades es crearan tantes tasques com particions, que s'executaran de forma distribuïda gràcies a PyCOMPSs.

Quan es distribueix una aplicació és essencial triar la granularitat de les tasques amb cura. En primer lloc, si la granularitat és molt petita, l'overhead de creació de tasques i de sincronització pot ser molt gran. En segon lloc, si la granularitat és massa gran, és possible que no sigui prou paral·lel. Aquests són coneixements previs que amb Hecuba els programadors no haurien de necessitar.

Aquest projecte aspira a implementar un gestor de la granularitat dinàmic integrat amb Hecuba i PyCOMPSs, amb el qual els programadors no hagin d'intervenir en la distribució de les dades, i tot i així aconseguint que les aplicacions siguin el més eficients possible.

Abstract

Hecuba is an interface and a set of tools written in Python and C++, developed by the Data-driven Scientific Computing team of the Barcelona Supercomputing Center. The objective is to provide programmers with an easy and efficient interaction with non-relational technologies, in this case, the distributed database Cassandra.

Hecuba can also be integrated with PyCOMPSs, a framework that facilitates the development and execution of applications in parallel in distributed infrastructures. This integration benefits from the distribution of the data that Hecuba offers through Cassandra, for its subsequent distributed computing with PyCOMPSs. To facilitate this distributed processing, another utility of Hecuba and the most important for this project is the data partitioner. This divides an object (which can be associated, for example, with a table in the database) into smaller ones that contain subsets of the data. **The term granularity in this project refers to the size of the created subobjects**. After dividing the data, as many tasks as partitions will be created, which will be executed in a distributed way thanks to PyCOMPSs.

When distributing an application, it is essential to choose the granularity of the tasks carefully. First of all, if the granularity is very small, the task creation and synchronization overhead can be very large. Second, if the granularity is too large, it may not be parallel enough. This is prior knowledge that programmers should not need with Hecuba.

This project aims to implement a dynamic granularity manager integrated with Hecuba and PyCOMPSs, with which the programmers do not have to intervene in the distribution of the data, and even so, making the applications as efficient as possible.

Agradecimientos

Después de cuatro meses, hoy escribo los agradecimientos que ponen punto y final a este trabajo de final de grado. Quiero agradecer a todas las personas que me han ayudado y apoyado durante este proyecto, que me ha aportado tanto a nivel profesional como personal.

Para comenzar, quiero agradecer a la directora del proyecto, Yolanda, por haberme guiado durante todo el proyecto. En este aspecto también quiero agradecer a todos mis compañeros de trabajo, quienes me han ayudado en muchos momentos y sin ellos este proyecto habría sido más complicado.

Además, este también es el fin de mi grado, por lo que quiero dar las gracias a todas las personas que me han acompañado durante estos cuatro años: Oleksandr, David, Alejandro, Raúl y Enric. Juntos hemos pasado muchos buenos y malos momentos que nos han unido inmensamente.

Para acabar, quiero agradecer a mis padres, por su apoyo incondicional durante el grado, y a mi pareja Maria Àngels, que ha sido mi mayor inspiración y motivación.

Muchas gracias a todos.

Tabla de contenidos

Lista de figuras

Lista de tablas

Glosario

BSC	**B**arcelona **S**upercomputing **C**enter
HPC	**H**igh **P**erformance **C**omputing
NoSQL	**N**o **SQL** o **N**ot **O**nly **SQL**
SQL	**S**tructured **Q**uery **L**anguage
UUID	**U**niversally **U**nique **Id**entifier
API	**A**pplication **P**rogramming **I**nterface

1. Contexto

1.1. Introducción

Cuando se trata con grandes conjuntos de datos y con una enorme carga de trabajo de consultas, las bases de datos NoSQL frecuentemente son una solución. De esta idea han surgido sistemas que ofrecen una gran escalabilidad y disponibilidad a costa de proporcionar solo un conjunto reducido de funcionalidades, obligando a los usuarios a ocuparse de lógicas complejas. Además, otra desventaja de estas bases de datos no relacionales es que para usarlas de manera eficiente hace falta tener conocimientos específicos de cada una.

Muchos científicos de diferentes ramas necesitan desarrollar aplicaciones con una entrada masiva de datos en entornos HPC, pero a la vez estos científicos no tienen los suficientes conocimientos de programación ni de uso de bases de datos no relacionales.

Como solución surgió la idea de Hecuba. Con Hecuba las aplicaciones pueden acceder a los datos como si fueran objetos normales guardados en memoria, y el código se traduce en tiempo de ejecución para utilizar los datos guardados en la base de datos. Gracias a esto, los programadores pueden almacenar los datos de forma distribuida como si programaran con los datos en memoria, es decir, el uso de la base de datos es totalmente transparente al usuario.

```
from hecuba import StorageDict
class MyData(StorageDict):
    '''
    @TypeSpec dict<<mykey:int>, mydata:str>
    '''

my_data = MyData()
my_data.make_persistent("mykeyspace.mytable")
my_data[0] = "one row"
```

Figura 1. Ejemplo de Hecuba

Como podemos observar en la Figura 1, para usar Cassandra con Hecuba en primer lugar tenemos que definir el modelo de datos. Para ello, hay que crear un objeto que herede de una clase de datos de Hecuba, en este caso el *StorageDict*, que implementa un diccionario persistente. En este ejemplo, el modelo de datos es una tabla con un entero como clave primaria y una columna que será de tipo texto. La Figura 2 muestra la tabla que se crea en Cassandra al usar este modelo de datos.

```
cqlsh> describe table mykeyspace.mytable ;

CREATE TABLE mykeyspace.mytable (
    mykey int PRIMARY KEY,
    mydata text
) WITH bloom_filter_fp_chance = 0.01
    AND caching = {'keys': 'ALL', 'rows_per_partition': 'NONE'}
    AND comment = ''
    AND compaction = {'class': 'org.apache.cassandra.db.compaction.SizeTieredCompactionStrategy', 'max_threshold': '32', 'min_threshold': '4'}
    AND compression = {'chunk_length_in_kb': '64', 'class': 'org.apache.cassandra.io.compress.LZ4Compressor'}
    AND crc_check_chance = 1.0
    AND dclocal_read_repair_chance = 0.1
    AND default_time_to_live = 0
    AND gc_grace_seconds = 864000
    AND max_index_interval = 2048
    AND memtable_flush_period_in_ms = 0
    AND min_index_interval = 128
    AND read_repair_chance = 0.0
    AND speculative_retry = '99PERCENTILE';
```

Figura 2. Ejemplo de modelo de datos

En segundo lugar, hay que crear un objeto de la clase previamente definida. Para que este objeto sea persistente, hay que llamar a la función *make_persistent* con los nombres del keyspace y la tabla deseados. A partir de aquí todos los datos que se inserten en el diccionario también irán a la tabla de Cassandra. Otra opción es insertar datos en memoria y después llamar a *make_persistent*, con lo que todos los datos del diccionario se insertarán también en la base de datos. Este método también se puede usar para instanciar objetos que ya existan en la base de datos. En la Figura 3 se puede ver cómo quedaría la tabla.

```
cqlsh> select * from mykeyspace.mytable ;

 mykey | mydata
-------+---------
     0 | one row
```

Figura 3. Ejemplo de inserción de datos

Actualmente, en la mayoría de situaciones los científicos utilizan ficheros como entrada de datos, lo que limita en gran medida la velocidad y la flexibilidad comparado con lo que puede ofrecer una base de datos. Gracias a Hecuba, pueden acceder a Cassandra de una forma fácil y eficiente.

El diagrama de flujo de cualquier aplicación que use Hecuba y distribuya la ejecución con PyCOMPSs es el siguiente:

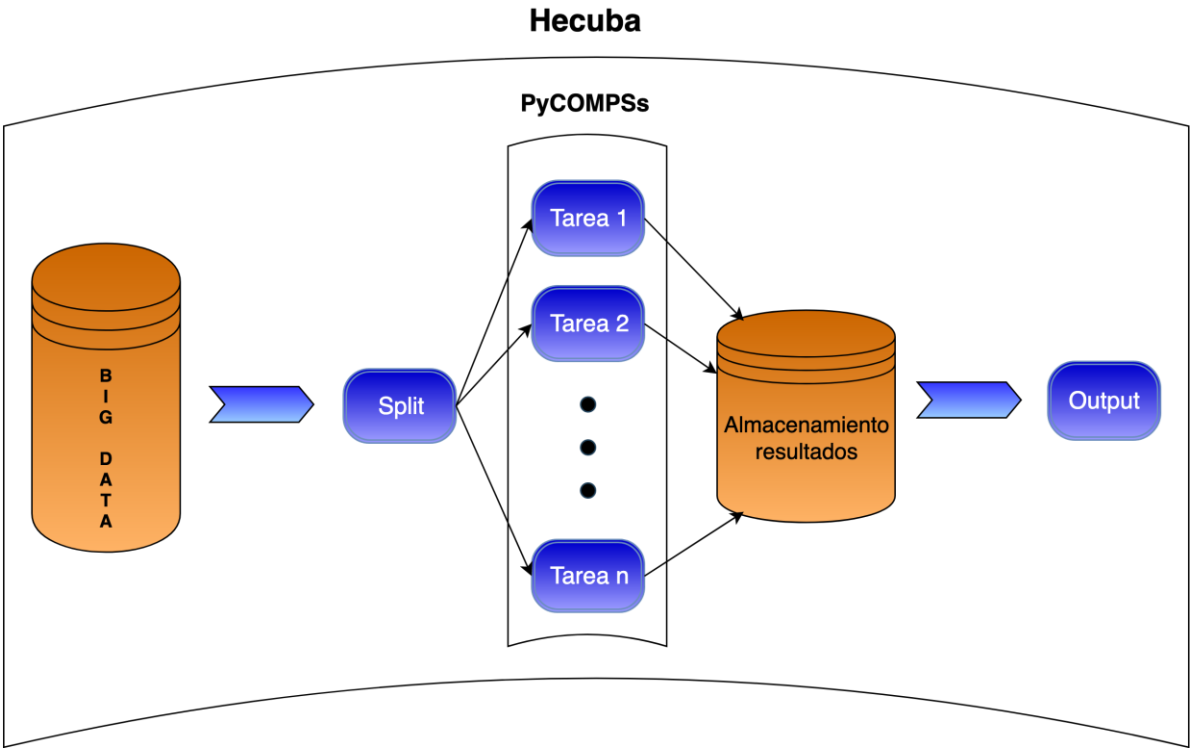


Figura 4. Diagrama de flujo de una aplicación

Como podemos ver en la Figura 4, partiendo de una base de datos y habiendo seguido los pasos anteriores, se puede realizar un *split* en los datos. Este s*plit* se refiere a dividir un objeto de Hecuba, que incluye todos los datos, en particiones que contienen cada uno un subconjunto de los datos. Cada una de las particiones será enviada a una tarea de PyCOMPSs, y serán procesadas en paralelo. La recolección de los resultados se puede hacer mediante la base de datos o mediante la misma aplicación, aunque en el caso de la imagen anterior se usa la base de datos. En el Anexo 1 podemos ver lo que sería la implementación del famoso recuento de palabras o *word count*.

Para facilitar el entendimiento del proyecto se van a definir diferentes conceptos relacionados con el mismo:

Keyspace: un keyspace es el objeto con nivel más alto de la base de datos que controla la replicación de los objetos que contiene. Los keyspaces contienen tablas, vistas materializadas, tipos definidos por usuarios, funciones y agregaciones.

Tabla: una tabla es donde se guardan los datos. Tienen filas y columnas, de manera similar a una tabla de una base de datos SQL.

Token: es un valor dado por una función de hash. Van desde -2^{63} a 2^{63}.

Rango de tokens: es un intervalo que comienza en un token y acaba en otro. Una fila se identifica en el clúster gracias a un token, y este token está asociado a un rango si está dentro del intervalo. Cada nodo tiene asociados varios rangos de tokens, con los que se decide a qué nodo irá cada fila, que es identificada por el token generado a partir de una función de hash de

la clave primaria. Esta distribución de los rangos de token en los diferentes nodos depende de la política de particionamiento que se esté usando. [4]

Partición: una partición es una porción de los datos. Cada partición tiene asignados diferentes rangos de tokens.

Tarea: en este contexto, una tarea será una función que se ejecutará de forma distribuida gracias a PyCOMPSs.

Master: el nodo donde se ejecuta el gestor de la granularidad.

Worker: nodo que ejecuta una o varias tareas.

Overhead: exceso de tiempo de, en este caso, computación, necesario para realizar una tarea específica.

Speedup: mejora de rendimiento.

Clave primaria: columna o combinación de columnas de la base de datos que identifica de forma única a cada fila de una tabla. Por lo tanto, no podrán existir dos filas en la misma tabla que se identifiquen por la misma clave primaria.

Partition key: parte de la clave primaria que sirve para determinar a qué nodo irán los datos, utilizando una función de hash.

Snapshot: copia de la base de datos en un momento determinado.

1.2. Formulación del problema

El problema principal de la actual implementación del particionador de datos es que, aunque el objetivo de Hecuba es que el programador no tenga que preocuparse por el almacenamiento ni la distribución de los datos, en realidad tiene que elegir el número de particiones de los datos que se crearán.

Si el programador no elige el número de particiones, por defecto se crearán 32, lo que quiere decir que después se crearán 32 tareas en paralelo. Si se deja este número por defecto lo más probable es que no se aproveche al máximo la paralelización, con lo cual el rendimiento de la aplicación puede verse severamente afectado.

A causa de esto, si los usuarios quieren implementar aplicaciones verdaderamente eficientes, necesitarán conocimientos específicos de programación paralela. Este hecho contradice totalmente el objetivo de Hecuba.

1.3. Objetivos

Por un lado, el objetivo principal del proyecto es diseñar e implementar un método de particionamiento de datos que decida la granularidad de las tareas. Se harán diferentes versiones de forma iterativa hasta alcanzar una solución que ofrezca un rendimiento

satisfactorio. Por lo tanto, el objetivo final del proyecto se alcanzará con la última versión, que deberá ser un método dinámico que tome decisiones en tiempo de ejecución.

Por otro lado, se pretende comparar el rendimiento que ofrece el método de particionamiento implementado, a partir de pruebas diseñadas de forma que muestren resultados significativos. Estas pruebas se realizarán para juzgar si el objetivo anteriormente mencionado se está cumpliendo, o se necesita hacer una versión con mejor rendimiento.

1.4. Actores implicados

En este apartado se definirán los actores implicados en este proyecto, es decir, a quién va dirigido el producto, quién lo usará y quién se beneficiará de sus resultados.

Barcelona Supercomputing Center

Este proyecto es realizado dentro del equipo Data-Driven Scientific Computing del Barcelona Supercomputing Center, por lo tanto, es el actor implicado principal. El equipo se encarga de desarrollar Hecuba, dónde irá integrado el gestor de la granularidad de las tareas, así que la mejora de rendimiento de la aplicación favorece a todos los integrantes del equipo.

Al mismo tiempo, el equipo Workflows and Distributed Computing también es una parte interesada, ya que son los desarrolladores de PyCOMPSs. Tanto los equipos de Hecuba como de PyCOMPSs suelen trabajar juntos para permitir su fácil integración y mejora.

Desarrollador

El desarrollador es la persona principal encargada de implementar, probar y documentar todo el software necesario. Además, es el responsable de la planificación temporal y de la gestión del proyecto, así como de cumplir las fechas límite.

Directora del proyecto

Es la principal responsable de guiar y aconsejar al desarrollador durante todo el proyecto. Asimismo, es la principal mánager del equipo Data-Driven Scientific Computing, de manera que está fuertemente relacionada con el proyecto.

Usuarios de Hecuba

La intención de Hecuba es facilitar la programación a investigadores de otras ramas, por lo que éstos se verán intensamente afectados por la mejora en simplicidad y rendimiento de Hecuba. Si este proyecto consigue buenos resultados los usuarios gastarán menos tiempo en paralelizar sus aplicaciones, lo que hará que su trabajo sea más eficiente.

Dado que la idea nace de científicos sin muchos conocimientos de programación, los principales usuarios de Hecuba son trabajadores o investigadores del Barcelona Supercomputing Center, además de socios en proyectos comunes.

2. Estado del arte

En esta sección se va a hablar sobre cómo se soluciona habitualmente este problema en la actualidad, y a estudiar si se puede aprovechar y adaptar una solución existente o si se tiene que diseñar una nueva.

Granularidad según el programador

En muchos casos, sobretodo en entornos de computación de altas prestaciones, la granularidad de las tareas es decidida por el mismo programador. De hecho, y como ha sido mencionado anteriormente, este es el actual funcionamiento de Hecuba.

Esto implica que posiblemente se consiga el mayor rendimiento, pero a la vez los programadores necesitan tener conocimientos de computación paralela. Como nosotros queremos simplificar el uso de Hecuba, esta no es una buena solución del proyecto.

Estrategia de MapReduce

MapReduce es el modelo de programación que da soporte a procesamiento de grandes cantidades de datos de forma paralela y distribuida más utilizado.

En este caso, al inicio de la ejecución el input es dividido en trozos de 16-64 MB (controlable por el usuario), para después iniciar varias copias del programa en un clúster de máquinas [5]. Por lo tanto, la granularidad se decide antes de iniciar la aplicación y no cambia en ningún momento de la ejecución.

Aun siendo el más utilizado, esta opción no es suficiente como solución del proyecto ya que se busca un gestor dinámico de la granularidad.

Otra solución

Hay una publicación titulada *Task granularity policies for deploying bag-of-task applications on global grids* [6] con información muy relevante para este proyecto.

En esta publicación se propone un planificador basado en dividir los lotes de datos en diferentes categorías, que tienen diferentes características cada una. Estas características son, por ejemplo, el tamaño del input o el tamaño del output. Al principio de la ejecución se calculan diferentes métricas para cada una de las categorías, como el tiempo de ejecución o el overhead. A continuación, la categoría que haya conseguido las mejores métricas (basado en un conjunto de políticas) será la que se utilice en el futuro. También propone que estas métricas se vayan actualizando durante la ejecución de la aplicación.

Esta es una muy buena solución, pero se sale del alcance del proyecto. Esto es porque utiliza algunas métricas que en el entorno de desarrollo no se pueden conseguir, como el tiempo de CPU o el overhead de una tarea, y otras que no tendrían sentido, como el tiempo máximo que una tarea puede estar ejecutándose en un recurso específico.

Aplicación a nuestro caso

Utilizando la estrategia de MapReduce se va a implementar una primera versión parecida para ofrecer más alternativas a los usuarios. Este será un particionamiento de las tareas con una granularidad de tamaño arbitrario, que podrá ser elegida por el programador.

Gracias a la publicación resumida en el apartado anterior, en este proyecto se pretende implementar una versión simplificada de este planificador, siendo cada una de las categorías una granularidad diferente. De la misma forma, al principio de la aplicación se medirá el tiempo de ejecución de las tareas con diferentes granularidades, y a lo largo de toda la ejecución se irá aproximando la granularidad ideal.

En conclusión, se va a aprovechar y a adaptar la opción de MapReduce puesto que es conveniente y sencilla, pero no suficiente. La parte más importante será la adaptación de la segunda publicación, que como se ha mencionado anteriormente, en su totalidad es demasiado compleja, pero se va a diseñar una solución nueva cogiendo las ideas base.

3. Alcance, metodología y planificación

3.1. Alcance

Fases del proyecto

El proyecto consta del diseño e implementación de diversas estrategias para determinar la granularidad de las tareas, seguido del testeo de éstas, por lo tanto, se han determinado tres fases bien diferenciadas:

1. **Análisis de las opciones y diseño de la implementación**. En esta fase habrá que entender el funcionamiento de Cassandra, Hecuba y PyCOMPSs para averiguar el mejor modo de implementar las funcionalidades requeridas.

2. **Implementación y testeo del software**. A partir del diseño realizado en la fase anterior se deberá implementar el software necesario. Además, se tendrán que realizar tests para corroborar que el software funciona correctamente.

3. **Pruebas de rendimiento**. Durante la fase se diseñarán y realizarán pruebas que midan el rendimiento de la implementación.

Durante la parte final del proyecto, las dos últimas fases se irán realizando de forma iterativa, para así implementar el software y poder comprobar su rendimiento. Gracias a las conclusiones de las pruebas, se procederá a implementar una nueva versión del software, que también necesitará ser probado. Este procedimiento se repetirá hasta alcanzar una solución satisfactoria.

Requisitos de los objetivos

Para que los objetivos sean alcanzados de forma satisfactoria al final del proyecto se deberán cumplir diferentes requisitos:

1. El software implementado mejora el rendimiento de la versión inicial de Hecuba.

2. Los usuarios no notarán que hay un gestor de la granularidad de las tareas detrás, solamente tendrán que elegir la estrategia del gestor.

3. El gestor dinámico aproxima rápidamente la granularidad ideal, de forma que prácticamente no se pierde tiempo realizando tareas con una granularidad que empeore el tiempo de ejecución total. Además, no se penaliza demasiado el tiempo de ejecución al estar corriendo continuamente.

El primer requisito es básico y fundamental para el proyecto, puesto que la idea del proyecto surge de que la versión inicial no es apropiada.

El segundo y tercer requisito son igual de importantes. Aunque el tercero se puede relajar según los obstáculos que surjan a lo largo del proyecto, el segundo deberá cumplirse siempre, si no, el objetivo de Hecuba de ser transparente a los usuarios se verá alterado.

Posibles obstáculos

La ejecución del gestor penaliza demasiado el tiempo total

Éste es un riesgo que podría ocurrir debido al gestor dinámico de la granularidad de las tareas, considerando que debe de estar ejecutándose en todo momento. En este caso habrá que plantear diferencias en la implementación.

Problemas durante la implementación

A causa de problemas en la implementación, ya sea por limitaciones del lenguaje Python, problemas de integración con PyCOMPSs, etc., es posible que se tenga que rediseñar la implementación del software.

Sobrecarga en el sistema de pruebas

Para las pruebas de rendimiento se va a usar en la medida de lo posible el computador MareNostrum 4, y debido a que este cuenta con un sistema de colas, hay posibilidades de que la cola esté saturada.

3.2. Metodología y rigor

Este proyecto necesita flexibilidad y resultados en poco tiempo, puesto que se plantea realizar una primera implementación seguida de pruebas de rendimiento y mejoras en el software de forma iterativa. Por esta naturaleza del proyecto sumado al poco tiempo que se tiene para realizar se ha decidido utilizar metodologías ágiles.

Para la aplicación de estas metodologías se utilizarán conocimientos adquiridos durante las asignaturas gestión de proyectos de software y proyecto de ingeniería del software, dónde se enseñan y se aplican.

Método de trabajo

Los diferentes principios que seguirá este proyecto se pueden observar en el Agile Manifesto [7].

Iteraciones cortas

Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale: Durante la implementación del software se realizarán iteraciones cortas debido a que puede haber problemas durante la misma, ocasionando cambios en el diseño. Cada iteración debe incluir partes del software necesario funcionando y testeado, y al llegar a la fase de pruebas de rendimiento, cada una debe incluir los resultados obtenidos de las pruebas y las mejoras realizadas.

Feedback continuo y cara a cara

The most efficient and effective method of conveying information to and within a development team is face-to-face conversation: Durante toda la realización del proyecto se irá presentando el avance del diseño y de la implementación a la directora del proyecto para que pueda ayudar a mejorar y a corregir errores. De la misma forma se recibirá feedback de los equipos de Hecuba y PyCOMPSs para facilitar el desarrollo y la integración del software, cara a cara siempre que sea posible.

Herramientas de seguimiento

GitHub

GitHub es la herramienta principal que se utilizará como control de versiones, con objetivo de facilitar la disponibilidad del código en cualquier momento y la recuperación de fallos. Se utilizará el repositorio actual de Hecuba[1] para que los demás compañeros del equipo puedan dar feedback al código fácilmente.

Métodos de validación

Para el seguimiento del proyecto se realizarán reuniones frecuentes con la directora del proyecto. Gracias a que la directora es la mánager del equipo dónde se realiza el proyecto estas reuniones serán facilitadas, y además podrá haber una comunicación constante para la rápida resolución de problemas y dudas.

Por otra parte, la principal herramienta de validación de software será el framework de tests unitarios unittest [8], disponible como un módulo de Python. Estos tests deberán ser exhaustivos para así validar el correcto funcionamiento de la solución.

Además, se usará la herramienta Travis CI [9], un servicio de integración continua de GitHub, para garantizar que todo el código que se genera pasa todos los tests. El funcionamiento se basa en que en el momento en el que se sube el código al repositorio, se crea un entorno remoto con las dependencias que se especifiquen en un fichero .yml, y seguidamente se ejecutan todos los tests. Gracias a esta herramienta, la integración del software del proyecto y de Hecuba será mucho más fácil, ya que si se detecta alguna incompatibilidad se verá reflejado al momento.

3.3. Planificación temporal

La duración estimada del proyecto es de aproximadamente cuatro meses. Contando como inicio el día 18 de febrero, hasta un día antes del comienzo de las presentaciones, el 1 de julio.

Descripción de las tareas

En este apartado se describirán las tareas planeadas en orden de su realización.

[1] GitHub - bsc-dd/hecuba: https://github.com/bsc-dd/hecuba.

Gestión del proyecto

Esta parte se refiere a todo lo relacionado con la asignatura de gestión de proyectos. Se puede dividir en cinco etapas con fechas definidas:

- Alcance y contextualización.
- Planificación temporal.
- Gestión económica y sostenibilidad.
- Documento final.
- Presentación oral.

Esta fase dura desde el 18 de febrero hasta el 29 de marzo.

Análisis de las opciones y diseño de la implementación

Esta tarea se basa en analizar el funcionamiento de Cassandra y de Hecuba para diseñar una implementación óptima, además de la integración con PyCOMPSs.

La mayor parte de la información asociada a Cassandra se obtendrá de la documentación proporcionada por Datastax [10], una empresa que distribuye y soporta una versión para empresas de Cassandra.

Acabado el análisis se procederá a diseñar la implementación del software necesario.

Implementación de un gestor de la granularidad por tamaño de datos

Ésta será la parte más básica del software del proyecto, la cual no será muy compleja, por lo que conllevará poco tiempo.

Este gestor sirve para proporcionar otra alternativa de particionamiento de datos a los usuarios de Hecuba.

Después de la implementación se crearán tests unitarios para comprobar el correcto funcionamiento del software. Por otra parte, también se harán tests en el clúster Minerva, ya que el objetivo del proyecto es que funcione de forma distribuida. De este clúster se hablará en el apartado 3.3. Planificación temporal – Recursos.

Implementación de un gestor dinámico de la granularidad

Esta tarea será la más larga y la que dará más trabajo, debido a que se implementará el software correspondiente al objetivo final del proyecto.

Este gestor se basará en que, al principio de la ejecución, se lanzarán tareas con diferentes granularidades para medir el tiempo que tardan en completarse. Seguidamente y durante toda la ejecución de las tareas, se irá aproximando la granularidad ideal.

De la misma forma que con el gestor por tamaño de datos, se crearán tests unitarios para comprobar el correcto funcionamiento del software, además de los tests en el clúster Minerva.

Esta tarea es un proceso iterativo en el que se harán sucesivas etapas de mejoras, en función de los resultados obtenidos en la tarea que se explicará a continuación.

Realización de tests de rendimiento

Se realizarán tests que den información útil acerca del rendimiento que ofrece el gestor dinámico. Estos tests deberán ayudar a sacar conclusiones acerca de dónde se está perdiendo rendimiento, para así poder mejorar la implementación.

Como se ha mencionado en la tarea anterior, estos tests se harán de forma iterativa junto con mejoras en el gestor dinámico, con la intención de conseguir el mejor rendimiento posible.

El entorno será, en la medida de lo posible, el supercomputador MareNostrum 4, para comprobar el rendimiento en un clúster de mayores dimensiones. Hay una desventaja respecto a Minerva, y es que MareNostrum 4 cuenta con un sistema de colas, por lo que será menos ágil.

Preparación de la defensa y lectura

Última tarea y con la que se dará por finalizado el proyecto. Entre 1 y el 5 de julio de 2019 se deberá defender el trabajo en sesión pública ante un tribunal de la especialidad, por lo tanto, se ha de preparar una presentación.

Diagrama de Gantt

En esta sección se han definido las subtareas para las tareas mencionadas anteriormente, para así poder asignarles una cantidad de horas específicas.

En el caso de la gestión del proyecto se han calculado unas dos horas al día de trabajo, mientras que para las demás se han tenido en cuenta cuatro horas al día. La Tabla 1 muestra la duración estimada de cada tarea.

Tarea	**Duración estimada (h)**
Gestión del proyecto	80
Análisis de las opciones y diseño de la implementación	56
Gestor por tamaño de los datos	56
Gestor dinámico	172
Tests de rendimiento	60
Preparación de la defensa	28
Total	452

Tabla 1. Tiempo estimado para cada tarea

A continuación, se resumen las tareas en un diagrama de Gantt:

Riesgo	Nombre de la tarea	Fecha de Inicio	Fecha final	Duración	Predecesoras
	Proyecto				
	Gestión del proyecto	**18/02/19**	**29/03/19**	**40d**	
	Alcance y contextualización	18/02/19	26/02/19	9d	
	Planificación temporal	27/02/19	04/03/19	6d	3
	Gestión económica y sostenibilidad	05/03/19	11/03/19	7d	4
	Documento final	12/03/19	20/03/19	9d	5
	Presentación oral	21/03/19	29/03/19	9d	6
	Análisis de las opciones y diseño de la implementación	**30/03/19**	**12/04/19**	**14d**	
	Análisis de las opciones	30/03/19	05/04/19	7d	
	Diseño de la implementación	06/04/19	12/04/19	7d	9
	Gestor por tamaño de los datos	**13/04/19**	**26/04/19**	**14d**	
	Implementación del software	13/04/19	19/04/19	7d	
	Tests unitarios	20/04/19	22/04/19	3d	12
	Tests clúster Minerva	23/04/19	26/04/19	4d	13
	Gestor dinámico	**27/04/19**	**23/06/19**	**58d**	
	Implementación del software	27/04/19	23/06/19	58d	
	Tests unitarios	18/05/19	23/06/19	37d	
	Tests clúster Minerva	21/05/19	23/06/19	34d	
	Tests de rendimiento	**25/05/19**	**23/06/19**	**30d**	
	Creación de los tests	25/05/19	28/05/19	4d	
	Realización de los tests en Marenostrum 4	29/05/19	23/06/19	26d	20
	Sacar conclusiones de los tests de rendimiento	29/05/19	23/06/19	26d	
	Preparación de la defensa	**24/06/19**	**30/06/19**	**7d**	
	Preparación de las diapositivas	24/06/19	27/06/19	4d	
	Preparación de la defensa	28/06/19	30/06/19	3d	24

Figura 5. Tareas definidas para el diagrama de Gantt

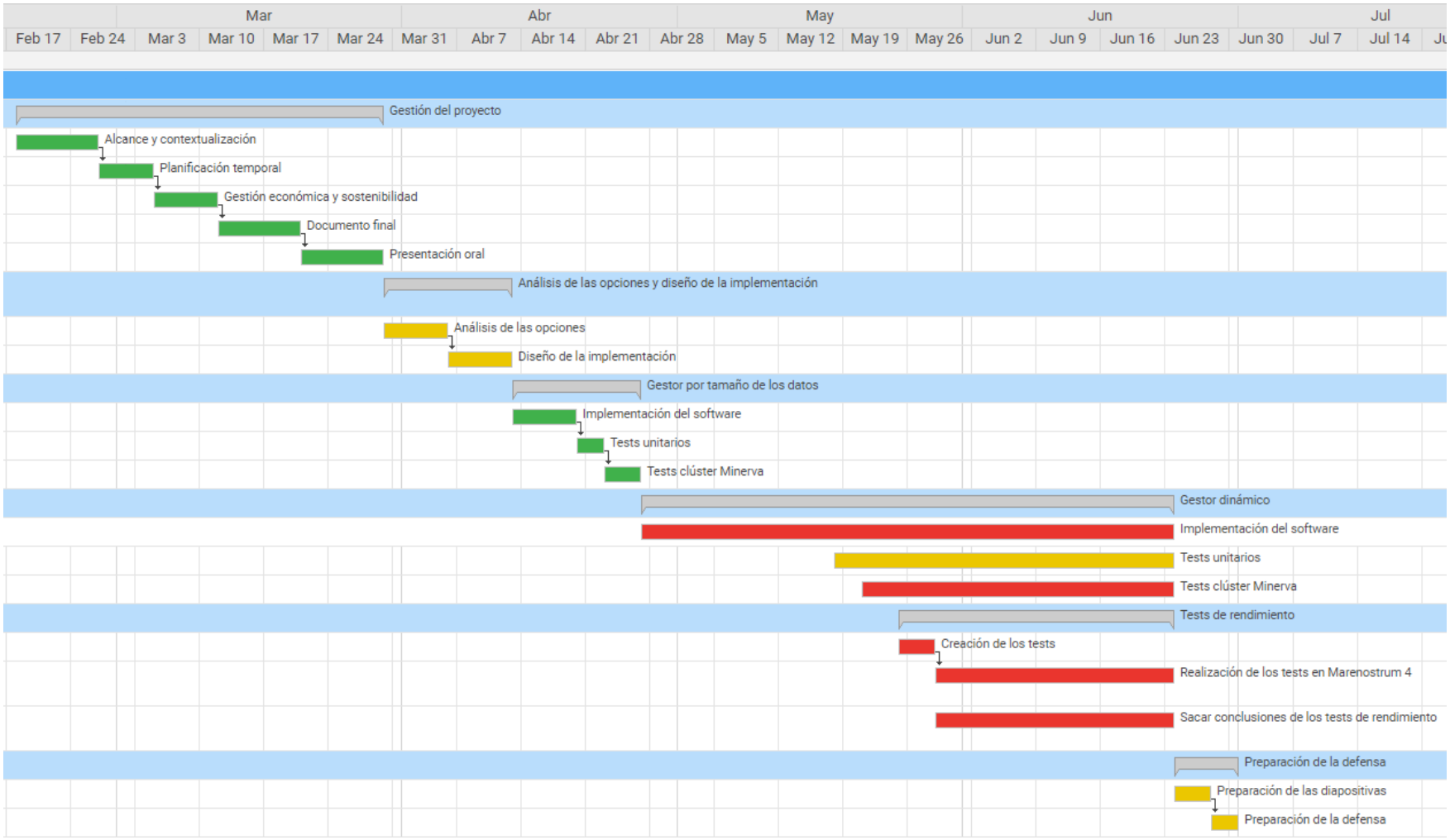


Figura 6. Diagrama de Gantt[2]

[2] Generado con https://app.smartsheet.com/.

Recursos

Los recursos que se han previsto para este proyecto se pueden dividir en recursos personales, recursos de software, recursos de hardware y recursos generales.

Recursos humanos

Una persona, el autor, que se dedicará a realizar toda la planificación y las tareas planteadas durante toda la duración del proyecto.

Recursos de software

- **Microsoft Word 2016**: para toda la documentación del proyecto.
- **Microsoft PowerPoint 2016**: para preparar las diapositivas de las presentaciones.
- **Git**: herramienta de control de versiones.
- **OpenSUSE Leap 15.0**: principal sistema operativo que se usará para implementar el proyecto.
- **Windows 10 Pro**: principal sistema operativo que se usará para documentar el proyecto.
- **PyCharm Professional 2018**: el entorno de desarrollo para la implementación del software.
- **Hecuba**: interfaz y conjunto de herramientas donde estará integrado el software del proyecto.
- **PyCOMPSs**: framework que facilita el desarrollo y ejecución de aplicaciones en infraestructuras distribuidas. El software del proyecto se verá beneficiado de algunas herramientas de PyCOMPSs, por lo que también tendrá que estar integrado.
- **Apache Cassandra**: base de datos distribuida que se utilizará durante todo el desarrollo. Esto es debido a que es el almacenamiento que usa Hecuba.

Recursos de hardware

- **Portátil Dell Latitude 7490**: ordenador que se utilizará durante todo el proyecto. Sus características son:
    - Procesador: 8x Intel Core i7-8650 CPU @ 1.90GHz.
    - Memoria: 16 GB de RAM.
    - Disco: 512GB SATA Class 20 Solid State Drive.
- **Clúster Minerva**: clúster del Departamento de Arquitectura de Computadores compuesto por cuatro nodos donde se comprobará el correcto funcionamiento del software. Al no tener un sistema de colas, se podrán ejecutar pruebas ágilmente.
- **Supercomputador MareNostrum 4**[3]: supercomputador del Barcelona Supercomputing Center que se usará para la realización de las pruebas de rendimiento. Cada nodo cuenta con

[3] Web con información sobre MareNostrum 4 | BSC-CNS: https://www.bsc.es/es/marenostrum/marenostrum.

dos sockets con 24 cores cada uno, por lo tanto, hay 48 cores por nodo. Además, se pueden utilizar hasta 50 nodos por persona, lo que hace un total de 2400 cores. Gracias a esto se podrán realizar numerosas pruebas de rendimiento, con la desventaja de tener un sistema de colas que puede ser bastante lento en ciertas ocasiones.

Recursos generales

- **Local**: se utilizará la oficina C6E201 del campus, donde el autor del proyecto trabaja.
- **Tasas municipales**: las tasas que el municipio de Barcelona indique.
- **Mobiliario**: solamente se necesitará una mesa.
- **Recursos de la oficina**: los recursos necesarios para cualquier inmueble: agua, electricidad, etc.

Alternativas y plan de acción

Como se mencionó en el apartado 3.2. Metodología y rigor, se van a utilizar metodologías ágiles. Gracias a la flexibilidad que ofrecen, permiten que la duración de las diferentes tareas varíe dependiendo de los posibles obstáculos que surjan. Por lo que, si una tarea se alarga por cualquier razón, la planificación se modificará para poder cumplir el plazo final. De la misma forma, si una tarea se acaba antes de lo esperado, la siguiente comenzará inmediatamente.

Estas desviaciones no afectarán demasiado al consumo de recursos materiales. En todo caso, los clústeres se usarán en mayor o menor medida, pero no afectarán de ninguna otra forma. Por otra parte, y debido a que cada tarea tiene asignados ciertos roles, sí que es posible que cambie la dedicación de los recursos humanos.

Además, al obtener feedback continuamente de la directora del proyecto, se podrán detectar rápidamente estas desviaciones en la planificación temporal, para así tomar las medidas necesarias.

De la tabla de tiempos estimados podemos ver que hay 80 horas para la gestión del proyecto y 372 horas para la realización. Como un crédito equivale a 25 horas, y el trabajo de final de grado cuenta con 3 créditos para la asignatura de gestión de proyectos, la duración de la gestión del proyecto debería de ser de 75 horas, aproximadamente el tiempo estimado en el apartado anterior. De la misma forma, el mismo proyecto son 15 créditos, lo que hace un total de 375 horas, otra vez es aproximadamente el tiempo estimado. Por lo tanto, la realización del proyecto es totalmente viable.

3.4. Presupuesto

En este apartado se va a realizar una estimación del presupuesto del proyecto. Para ello se va a dividir en dos secciones, estas son: costes directos e indirectos. A su vez, los costes directos se dividirán en tres teniendo en cuenta los tipos de recursos mencionados en el apartado 3.3. Planificación temporal: recursos humanos, recursos de software y recursos de hardware. Para

acabar, los costes indirectos son los que no se pueden asignar directamente al software producido.

Para el cálculo de las estimaciones se ha tenido en cuenta que la duración del proyecto es de cuatro meses.

Costes directos

Recursos humanos

La siguiente tabla muestra los costes por hora de cada uno de los recursos humanos que se necesitan para el proyecto[4]:

Rol	**Coste por hora (€/h)**
Gestor de proyectos software (G)	20
Analista de software (A)	14
Desarrollador de software (D)	13
Tester de software (T)	10

Tabla 2. Coste por hora estimado para cada rol

A fin de estimar los costes totales, podemos dividir los roles según las tareas que realizarán, de esta forma será más fácil estimar el coste total de cada rol:

Tarea	**Horas (h)**	**Dedicación por rol (h)**			
		G	**A**	**D**	**T**
Gestión del proyecto	80	80			
Análisis de las opciones y diseño de la implementación	56		56		
Gestor por tamaño de los datos	56			42	14
Gestor dinámico	172			114	58
Tests de rendimiento	60				60
Preparación de la defensa	28	28			
Total	**452**	**108**	**56**	**156**	**132**

Tabla 3. Estimación de horas por rol

[4] La estimación coste por hora es sacada de la web www.payscale.com/research/ES.

A partir de las dos tablas anteriores podemos estimar el coste total de recursos humanos:

Rol	**Horas (h)**	**Coste total (€)**
Gestor de proyectos software	108	2.160
Analista de software	56	784
Desarrollador de software	156	2.028
Tester de software	132	1.320
Total	**452**	**6.292**

Tabla 4. Presupuesto de recursos humanos

Recursos de software

La siguiente tabla muestra los costes totales de software que se imputan a nuestro proyecto:

Recurso	**Coste total (€)**	**Vida útil (años)**	**Amortización (€)**
Microsoft Office Word/PowerPoint 2016[5]	229,99	3	25,56
Git	0,00	3	0,00
OpenSUSE Leap 15.0	0,00	3	0,00
Windows 10 Pro	199,99	3	22,23
PyCharm Professional 2018[6]	199,00	1	66,34
Hecuba	0,00	3	0,00
PyCOMPSs	0,00	3	0,00
Apache Cassandra	0,00	3	0,00
Total	**628,98**		**114,13**

Tabla 5. Presupuesto de software

[5] Coste sacado de la tienda oficial de Microsoft: www.microsoft.com.
[6] Coste sacado de la tienda oficial de JetBrains: www.jetbrains.com.

Recursos de hardware

La siguiente tabla muestra los costes totales de hardware que se imputan a nuestro proyecto:

Recurso	**Coste total (€)**	**Vida útil (años)**	**Amortización (€)**
Portátil Dell Latitude 7490[7]	2.613,85	4	217,83
Clúster Minerva	0,00	4	0,00
Supercomputador MareNostrum 4	0,00	4	0,00
Total	**2.613,85**		**217,83**

Tabla 6. Presupuesto de hardware

El coste de Minerva es de 0,00 € porque es un hardware obsoleto que ya está amortizado. Además, el coste de MareNostrum 4 también es de 0,00 € porque se usará de forma muy puntual, por lo tanto, es negligible.

Costes indirectos

La siguiente tabla muestra los costes indirectos del proyecto:

Recurso	**Coste total (€)**
Alquiler oficina	5000,00
Mobiliario oficina	1500,00
Gastos oficina (agua, electricidad, tasas municipales, etc.)	3000,00
Total	**8500,00**

Tabla 7. Presupuesto de costes generales

[7] Coste sacado de la tienda oficial de Dell: www.dell.com.

Presupuesto total

Juntando los presupuestos estimados en los apartados anteriores podemos ver el presupuesto total del proyecto:

Concepto	**Presupuesto (€)**
Recursos humanos	6.292,00
Recursos de software	114,13
Recursos de hardware	217,83
Gastos generales	8.500,00
Imprevistos	1.000,00
Total	**16.123,96**

Tabla 8. Presupuesto total del proyecto

Para acabar, el cálculo del presupuesto total contando impuestos (21%) es de **19.510,00 €**. Además, se han añadido mil euros para gastos de imprevistos.

Control de presupuesto

Es posible que el presupuesto del proyecto tenga que modificarse a causa de obstáculos durante su realización. Sin embargo, no todos los presupuestos tienen la misma probabilidad de que necesiten ser modificados.

El presupuesto de recursos humanos es el más probable que sufra cambios, dado que se usarán metodologías ágiles y la duración de las tareas puede verse alterada. Por ello, es posible que algún rol necesite más horas de las estimadas para una tarea, lo que también comportaría una reducción de tiempo de otra tarea. Debido a esto, el tiempo de trabajo de los diferentes roles se vería afectado, así que el presupuesto también ya que cada rol tiene un coste por hora diferente.

Sin embargo, la probabilidad de que el presupuesto de recursos de software cambie es prácticamente nula. Hecuba es totalmente gratuito, por lo que el código que se desarrolle para el proyecto no podrá usar ningún tipo de librerías de pago. Por lo demás, el software mencionado es el único necesario.

Con relación al presupuesto de hardware, la única posibilidad de que sea modificado es que el portátil que se planea utilizar se vea inutilizado por alguna avería o virus. En este caso, el portátil se sustituirá por otro con un coste mucho menor mientras el primero se arregla.

Por otra parte, los clústeres Minerva y MareNostrum son proporcionados por otros agentes, y la probabilidad de que ninguno de los dos se pueda usar durante toda la duración del proyecto es casi nula. Con estos clústeres ya habrá suficiente para la realización del proyecto y no se necesitará más hardware.

En cuanto a los gastos de recursos generales, estos son bastante orientativos, así que es posible que requieran alguna modificación en el futuro.

Para solventar los cambios de presupuesto debido a desviaciones se ha añadido un presupuesto para imprevistos, que será suficiente para asegurar que el aumento en los costes de recursos humanos o generales no implica un gasto fuera de lo previsto.

Viabilidad económica

Discusión de la viabilidad económica

Este proyecto es parte de Hecuba, el cual es un proyecto open source. Por lo tanto, no tiene sentido calcular el VAN puesto que no va a tener ventas. Sin embargo, se va a justificar en qué puede beneficiar a la empresa.

La solución de este proyecto puede aportar numerosos beneficios. El principal contribuye al objetivo de Hecuba, que es mejorar la productividad de los trabajadores. La programación será mucho más eficiente sin necesidad de tener unos conocimientos que antes habrían sido necesarios. Además, el rendimiento de las aplicaciones puede mejorar drásticamente debido a la mejor explotación de los recursos.

Gracias a esto, otros centros de investigación se pueden ver atraídos por Hecuba y que empiecen a usarlo. Como consecuencia, la participación de Hecuba en proyectos europeos aumentaría, recibiendo más financiación de la Comisión Europea.

En conclusión, la realización del proyecto beneficia claramente a la empresa, tanto a corto plazo, gracias a la mejora de la productividad de los trabajadores, como a largo, gracias a la posible expansión del uso de Hecuba.

Impacto económico

El sistema será mantenido por el propio autor del proyecto y será sostenido económicamente por el Barcelona Supercomputing Center. Esto no implica ningún tipo de gasto adicional, puesto que una vez acabado el proyecto su mantenimiento no tendrá coste.

3.5. Sostenibilidad

En este apartado se va a presentar un informe de sostenibilidad del proyecto desde tres puntos de vista: el ambiental, el económico y el social.

Sostenibilidad económica

En el apartado 3.4. Presupuesto, ya se han estimado los costes de realización del proyecto, incluyendo los recursos humanos y materiales.

Como se ha mencionado anteriormente, hay poca probabilidad de que el presupuesto del proyecto varíe aparte del de recursos humanos. Además, debido a que el resultado del proyecto

es un software, una vez esté acabado no necesitará ningún tipo de gasto a no ser que en el futuro se decida realizar alguna actualización.

En cuanto a si el proyecto se podría haber realizado con un presupuesto menor, difícilmente esto podría ser posible. En primer lugar, el gasto de recursos humanos es necesario a no ser que el tiempo de realización del proyecto resulte ser menor del estimado. En segundo lugar, podría haberse usado la versión gratis de PyCharm, pero eso probablemente hubiera tenido consecuencias en otros aspectos. El resto del software de pago es totalmente necesario. En cuanto a hardware, el único coste es el portátil. Este podría haber sido más económico, pero el hecho de tener un disco de estado sólido y 16 GB de RAM, agiliza la realización del proyecto. Esto es debido a que hay que tener ejecutándose el entorno de desarrollo PyCharm además de Cassandra para realizar pruebas, lo que pide bastantes recursos.

Actualmente, el problema que se quiere abordar se resuelve fijando una granularidad arbitraria en la mayoría de los casos, lo que puede provocar que las aplicaciones distribuidas sean más lentas de lo que podrían ser. En otros casos, el desarrollador tiene que decidir la granularidad, y hace que se gaste más tiempo del necesario en el desarrollo. La solución propuesta en el proyecto apunta a resolver ambos casos, por lo tanto, mejora claramente a las existentes. Si el software del proyecto alcanza los objetivos, el rendimiento de las aplicaciones mejorará, y un menor tiempo de ejecución implica menos presupuesto. También agilizará el desarrollo de aplicaciones, lo que implica un menor tiempo de trabajo y de la misma forma, un menor presupuesto.

Sostenibilidad social

Este proyecto se va a realizar dentro de un centro de investigación, por lo cual al trabajar en este entorno durante su realización me ayudará a decidir si quiero dedicar mi carrera a la investigación. De igual manera, me ayudará a decidir si quiero continuar especializándome en los ámbitos del Big Data y la computación distribuida.

En la mayoría de casos, el problema actualmente se resuelve siendo el programador quien decide la granularidad de las tareas, ya que es un entorno científico en el que se necesita el mayor rendimiento posible. Gracias al proyecto, los usuarios de Hecuba tendrán ventajas respecto a este problema. Estos no necesitarán tener tantos conocimientos de computación distribuida, y aun así verán una mejora en el rendimiento de sus aplicaciones. Al dedicar menos tiempo al desarrollo y ejecución de las aplicaciones su investigación se verá impulsada, por lo que la solución del proyecto mejorará socialmente a las existentes.

Sin embargo, no hay una necesidad social del proyecto, puesto que la cantidad de potenciales usuarios del mismo es bastante reducida. A pesar de esto, si el rendimiento de las aplicaciones mejora mucho gracias a este proyecto, es posible que el uso de Hecuba se extienda a muchos otros centros de investigación. De cualquier forma, se intenta mejorar la calidad de vida de sus usuarios.

Sostenibilidad ambiental

Se va a hablar de la sostenibilidad ambiental teniendo en cuenta los recursos que se van a utilizar, que ya han sido explicados en el apartado 3.3. Planificación temporal – Recursos..

Siempre que se esté trabajando en el proyecto se va a hacer en el portátil, por lo tanto, hay que examinar su gasto energético. Según sus especificaciones, consume una media de 60 Whr. Dado que la duración del proyecto es de 452 horas, esto hace un total de 27120 W, y a su vez, 10441 kg de CO_2 aproximadamente. Este consumo energético es algo normal y necesario debido a su continuo uso.

En muchas ocasiones se van a utilizar los clústeres Minerva y MareNostrum 4. Solo podemos saber el gasto pico de MareNostrum, siendo de 33,7 kW, una gran cantidad. Con únicamente este dato no podemos saber el consumo energético de nuestro proyecto en lo que respecta al uso de los clústeres, dado que no sabemos ni la cantidad de nodos ni el tiempo que se usarán. Por ahora y dado que la intención es utilizarlo de forma muy puntual, podemos asumir que el consumo es negligible.

Aunque no sepamos el gasto exacto, podemos decir que el impacto ambiental de Minerva es mucho menor que si se utilizara un clúster propio de las mismas dimensiones, ya que es un clúster obsoleto que se está reutilizando.

Para acabar, y debido a que el objetivo del software del proyecto es reducir el tiempo de desarrollo y de ejecución de aplicaciones científicas, el consumo energético de los usuarios se verá reducido. Esta reducción puede ser de gran cantidad, puesto que estas aplicaciones se suelen ejecutar en entornos de computación de alto rendimiento. Gracias a esto, la solución del proyecto es notablemente mejor ambientalmente que las soluciones existentes.

4. Background

Antes de explicar los gestores de la granularidad implementados, es importante comprender cómo se van a particionar los datos según la granularidad escogida.

Como se ha dicho en la introducción, con Hecuba se puede acceder a los datos como si fueran objetos normales guardados en memoria. Para conseguir esto, cada objeto está asociado a un keyspace, a una tabla y a unos rangos de tokens. A la hora de crear particiones de un objeto, las particiones tendrán un subconjunto de los rangos de tokens del objeto original. Esta manera de dividir los datos entre objetos es la misma que usa Cassandra para distribuir los datos entre los nodos, llamada *consistent hashing* [11].

En la Figura 7 se puede ver un ejemplo de su funcionamiento simplificado. Primero de todo, cada nodo tiene asociado un rango de tokens. Cuando se insertan datos en el clúster, el primer paso es aplicar una función hash a la partition key, generando un token asociado a esos datos. Ese valor de token estará dentro del rango de tokens propiedad de un nodo, al que irán los datos. [12]

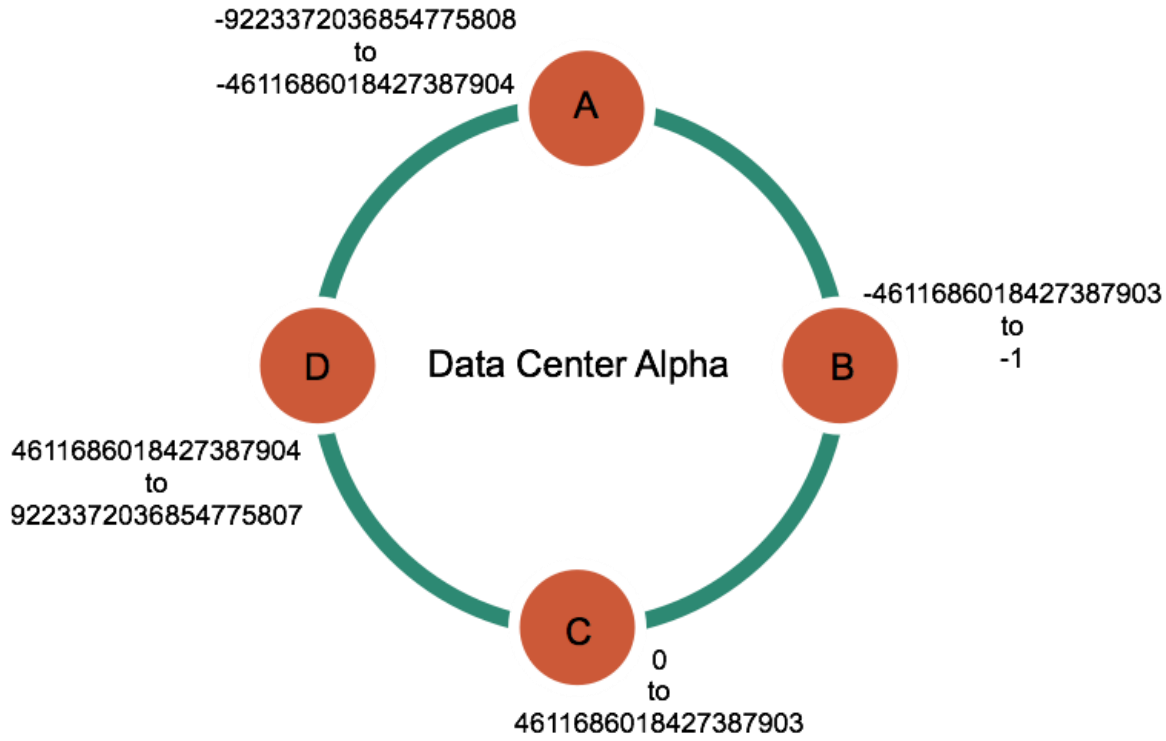


Figura 7. Tokens en un clúster de cuatro nodos [13]

Siguiendo este ejemplo, un objeto de Hecuba sin particionar tendrá asociados unos rangos de tokens que cubran todos los datos, en este caso sería de la siguiente manera:

$$\textbf{\textit{tokens}} = [\ (-9223372036854775808, token_1), \ldots (token_{n-2}, 9223372036854775807)\ ]$$

Sin embargo, si dividimos este objeto en cuatro particiones, sus rangos de tokens podrían ser:

partition1 = [(-9223372036854775808, $token_1$), … ($token_{n-2}$, -4611686018427387904)]

partition2 = [(-4611686018427387904, $token_1$), … ($token_{n-2}$, -1)]

partition3 = [(0, $token_1$), … ($token_{n-2}$, 4611686018427387903)]

partition4 = [(4611686018427387904, $token_1$), … ($token_{n-2}$, 9223372036854775807)]

De este modo, cada objeto tiene asociados los datos de un nodo, y pueden ser procesados en paralelo.

Por lo tanto, para crear particiones de objetos en Hecuba lo único que hay que hacer es dividir los rangos de tokens y asociarlos a nuevos objetos con las mismas características que el original. Para dividir los rangos de tokens se hace siguiendo estos pasos (ver Anexo 2 para el código completo):

1. En el entorno de Hecuba hay una variable definida de tal forma:

   min_number_of_tokens: mínimo número de tokens que debe haber en total.

2. Si la cantidad de tokens del objeto a dividir es menor que el mínimo número de tokens, hay que crear nuevos rangos.

   2.1. En este caso, la cantidad de tokens por partición será:

   if len(self._father.tokens) < min_number_of_tokens:
   tokens_per_partition = min_number_of_tokens / number_of_partitions

   2.2. Partiendo del primer token, y hasta que haya tantos tokens como *tokens_per_partition*, se van añadiendo pares de tokens con una amplitud de:

   $((2^{64}) - 1) / min\_number\_of\_tokens$

   2.3. Se hace *yield* de la lista de tokens creados.

3. Si el tamaño de tokens del objeto a dividir es mayor o igual que el mínimo número de tokens, entonces podemos dividir los rangos de tokens entre las particiones.

   3.1. En este caso, la cantidad de tokens por partición es:

   tokens_per_partition = max(len(father._tokens) / number_of_partitions, 1)

   3.2. Se hace *yield* de un trozo de la lista de tokens de tamaño *tokens_per_partition.*

Cuando se itera sobre esta función, gracias a la palabra clave *yield*, se devolverá el rango de tokens que se acaba de definir, y la siguiente vez que se itere continuará la ejecución a partir de donde se hizo *yield* por última vez.

Esto da resultado a un conjunto de particiones de los datos, que son los subobjetos creados a partir de otro objeto, y contienen un subconjunto de los datos. Podemos entender estas

particiones como lotes de datos que se procesarán en paralelo. La agregación de los datos de todas las particiones es la totalidad de los datos del objeto original, y cada una de estas particiones será procesada de forma distribuida gracias a PyCOMPSs.

Para acabar, es muy importante entender que **cuando se menciona el término granularidad nos referimos al valor que toma *tokens_per_partition*, ya que esta determinará la cantidad de datos asociados a una partición**. **Esta variable depende de *number_of_partitions*, que será la que se irá modificando en tiempo de ejecución para poder determinar la granularidad de los objetos**, y así se consigue que el comportamiento del gestor no se vea alterado por el valor de *min_number_of_tokens* ni por la cantidad de tokens del clúster.

5. Gestores de la granularidad

5.1. Introducción

En este apartado se van a explicar los dos gestores de la granularidad implementados, el gestor estático por tamaño de los datos y el gestor dinámico. Añadidos al gestor simple original, en total son tres gestores de la granularidad disponibles en Hecuba. Estos se pueden utilizar conjuntamente en las aplicaciones sin ningún tipo de problema, ya que el gestor que se va a utilizar se decide justo antes de iniciar el particionamiento de los datos, y se mantiene durante toda la ejecución de las tareas asociadas a ese y solo ese particionamiento. Por lo tanto, en una misma aplicación podemos estar ejecutando dos tipos de tareas diferentes, con diferentes gestores de la granularidad, en paralelo. La Figura 8 es un ejemplo de cómo podríamos usar los tres gestores en paralelo sin ningún problema.

```
# Gestor original
config.partition_strategy = "SIMPLE"
for partition1 in data1.split():
    task1(partition1)

# Gestor estático por tamaño de los datos
config.partition_strategy = "TABLE_SIZE"
config.optimal_partition_size = 64  # in kb
for partition2 in data2.split():
    task2(partition2)

# Gestor dinámico
config.partition_strategy = "DYNAMIC"
for partition3 in data1.split():
    task2(partition3)

compss_barrier()    # bloquea hasta que todas las tareas acaben
```

Figura 8. Ejemplo de uso de los gestores de la granularidad

5.2. Gestor estático por tamaño de los datos

Introducción

En este capítulo se va a describir la primera solución a este problema, el gestor estático de la granularidad por tamaño de los datos. Como se ha mencionado anteriormente, esta solución no es suficiente como solución del proyecto, pero se ha implementado para dar más opciones a los usuarios. La ventaja de este gestor es que no añade overhead, por lo que si el usuario sabe de entrada cuál es la granularidad ideal, puede usarlo directamente.

Este gestor de la granularidad es similar al que hay inicialmente, en el que el usuario escogía la cantidad de particiones de los datos que se hacían. En este caso, el usuario lo que tiene que elegir es el tamaño de las mismas particiones, de manera que la cantidad que se crearán será proporcional al tamaño de los datos.

Esta granularidad se escoge al principio y no cambia durante toda la ejecución de las tareas, por lo tanto, es un gestor estático de la granularidad.

Diseño e implementación

Antes de dividir los datos, el usuario debe definir cuál es el tamaño que quiere para las tareas, en kilobytes, para que el gestor pueda dividir los datos en lotes de este tamaño.

En este momento el usuario ya puede utilizar el particionador de datos. Al principio de la ejecución del particionador, se necesita saber el tamaño total de los datos que se quieren dividir, sin embargo, hay que realizar unos pasos previos para poder calcular bien este tamaño.

En primer lugar, es necesario hacer *flush* de los datos, lo que quiere decir que se fuerza la escritura de los datos en disco. Esto es debido a que cuando ocurre una escritura, Cassandra guarda los datos en una estructura de memoria llamada *memtable*, como podemos ver en la Figura 9. La *memtable* es una caché con política de escritura write-back, lo que quiere decir que es posible que haya datos en la caché que no estén en disco, a no ser que se haga un *flush* de los datos [14]. Para ello se ha utilizado la herramienta de Cassandra *nodetool* [15], una interfaz de línea de comandos para gestión de clústeres. Simplemente ejecutando el comando *nodetool flush* se consigue este objetivo.

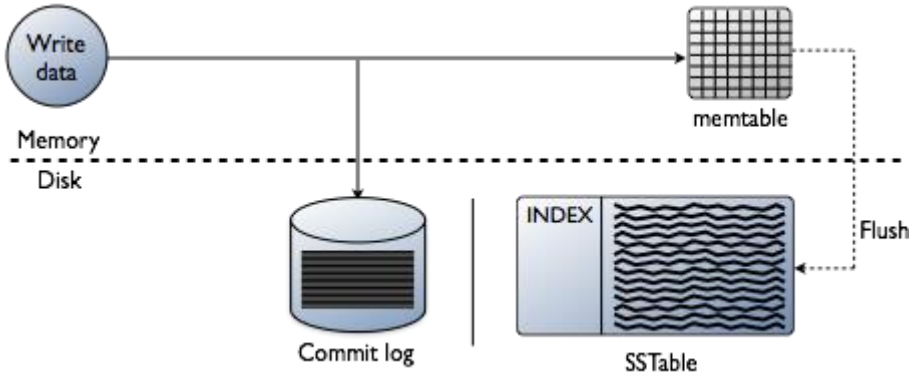


Figura 9. Estructuras de persistencia de Cassandra

En segundo lugar, Cassandra tiene una tabla de sistema llamada *system.size_estimates*, que nos ofrece información sobre las particiones en diferentes tablas. No obstante, cuando se insertan o se truncan grandes cantidades de datos es posible que estas estimaciones se vuelvan incorrectas. Para actualizar esta tabla con los datos reales, solamente se tiene que ejecutar el comando *nodetool refreshsizeestimates*, con la misma herramienta utilizada anteriormente.

Como se ha mencionado antes, para medir el tamaño total de los datos es suficiente con consultar la tabla *system.size_estimates*. Las columnas de esta tabla son:

- **keyspace_name**: keyspace de la tabla.
- **table_name**: nombre de la tabla.
- **range_start**: inicio del rango de tokens.
- **range_end**: final del rango de tokens
- **mean_partition_size**: tamaño medio de las particiones del rango de tokens en la tabla.
- **partitions_count**: número de particiones del rango de tokens en la tabla.

Los dos primeros parámetros sirven para obtener solamente la información que nos interesa, mientras que los dos últimos nos sirven para hacer el cálculo del tamaño total de los datos. Debido a que cada tabla tiene diferentes rangos de tokens, es necesario hacer una agregación:

```
for mean_partition_size, partitions_count in table_information:
    total_size_bytes += mean_partition_size * partitions_count
```

El tamaño total en bytes es agregar, por cada fila de la tabla *system.size_estimates*, el producto del tamaño medio de las particiones por el número de particiones.

Ahora que ya tenemos el tamaño total de la tabla, podemos calcular el número de particiones que se crearán de la siguiente forma:

```
number_of_partitions = ceil(float(table_size) / float(optimal_partition_size))
```

Para obtener el número de particiones se divide el tamaño de la tabla entre el tamaño óptimo de una partición, que fue definido por el usuario anteriormente, redondeando hacia arriba.

Ahora que ya tenemos el número de particiones, se puede proceder a dividir los datos de la misma manera que se ha explicado en el apartado 4. Background. Este gestor solamente calcula el número de particiones, y cuando lo ha elegido, el gestor pasa a ser el original.

Para ver el código completo ir al Anexo 3.

Conclusiones

Como se ha mencionado en la introducción, este gestor de la granularidad es muy similar al original, ya que el número de particiones que se crearán vendrá dado por el programador. Sin embargo, este ofrece una funcionalidad extra. A diferencia del gestor original, la cantidad de particiones no va a ser constante, sino que se verá alterada por el tamaño de los datos de entrada.

Aun así, esto no es suficiente como solución del proyecto, ya que se busca que el uso de la base de datos y la distribución de la computación sea totalmente transparente al usuario. Algunas aplicaciones pueden verse beneficiadas de este gestor de la granularidad, pero también requiere que los usuarios escojan dicha granularidad. De nuevo, el hecho de que los usuarios tengan que decidir la granularidad de las tareas implica que necesiten conocimientos de programación

distribuida. Como nosotros queremos un gestor de la granularidad con el que el usuario no tenga que tomar ninguna decisión, en el siguiente apartado se explicará la solución propuesta.

5.3. Gestor dinámico

Introducción

En este capítulo se va a describir el proceso realizando durante la implementación y las pruebas de rendimiento del gestor dinámico de la granularidad. Se va a proceder de la forma explicada en la planificación: se realizarán diferentes versiones de este gestor de forma iterativa, de manera que se harán experimentos en MareNostrum 4 para comprobar el rendimiento de cada una de las versiones y se elegirá la que ofrezca unas mejores métricas. Estas versiones son incrementales, con lo que cada versión trata de mejorar la anterior.

Este gestor de la granularidad se pretende que sea dinámico, es decir, que vaya aproximando la granularidad ideal durante toda la ejecución de la aplicación. Además, se van a mostrar gráficas que comparen el rendimiento de las diferentes versiones para poder llegar a una conclusión.

Versión 1[8]

Diseño e implementación

El diseño de la primera versión se basa en elegir la granularidad que ofrece un mejor rendimiento a partir de métricas que se consigan en tiempo de ejecución. Al principio, se van a lanzar una lista de tareas con diferentes granularidades. Cada vez que acabe una tarea, el master recoge el tiempo total de ejecución de la misma. A partir de este tiempo, se computa una métrica del tipo *tiempo / cantidad de datos*, para calcular el rendimiento (más adelante se explicará con más detalle). La granularidad que haya conseguido un rendimiento mejor, será la que se usará para procesar todos los datos restantes.

En primer lugar, si queremos determinar cuál es la tarea con un mejor rendimiento en tiempo de ejecución, necesitamos saber su tiempo de ejecución. Para ello se ha usado el API de PyCOMPSs para Hecuba, que dispone de unas funciones que se ejecutan al inicio y al final de las tareas. Gracias a esto, los workers pueden guardar en la base de datos el tiempo al empezar y al acabar las tareas, que después el gestor utilizará para calcular el tiempo total. Estos tiempos se guardarán en una tabla llamada *hecuba.partitioning*, cuyas sus columnas se han definido de la siguiente forma:

[8] Código de la primera versión: https://github.com/adrianespejo/Dynamic-granularity-manager/blob/master/partitioners/partitioner1.py.

- **storage_id (uuid)**: UUID que representa un objeto de Hecuba. En este caso, será una de las particiones de los datos. Hace de clave primaria, debido a que en los workers solo se puede saber este parámetro.
- **start_time (double)**: tiempo al inicio de la ejecución de la tarea.
- **end_time (double)**: tiempo al final de la ejecución de la tarea.
- **number_of_partitions (int)**: valor que se ha utilizado para determinar la cantidad de tokens de la partición.
- **partitioning_uuid (uuid)**: UUID que representa el particionamiento de datos que está en ejecución. Sirve para diferenciarse de otros procesos que se estén ejecutando en el mismo clúster.

Además, para que la tabla no se llene demasiado, se ha fijado un *default_time_to_live* de un día para toda la tabla. Esto hará que las filas correspondientes a particionados antiguos se eliminen al pasar un día.

El funcionamiento de la primera versión del gestor dinámico de la granularidad es el siguiente:

Al inicio del particionamiento, se lanzan tantas tareas como nodos disponibles. Para estas tareas, su granularidad se decide modificando la variable *number_of_partitions*, tal y como se explicó al final del apartado 4. Background.

1. Los números de particiones se escogen de una lista con opciones habituales en nuestro entorno de trabajo, cogiendo tantas como nodos disponibles.
2. Al escoger una granularidad, se crea una partición. En este momento, se insertan sus datos en la tabla *hecuba.partitioning*, salvo el tiempo de inicio y de fin. Estos son: *storage_id, number_of_partitions y partitioning_id*. Se insertan ahora porque el worker solo podrá saber el *storage_id*, y en el futuro el master necesitará estos datos.
3. El gestor realiza una espera mientras que no acaben las tareas iniciales. Esto quiere decir que, mientras los campos *start_time* y *end_time* de todas sigan a *null*, el hilo del gestor en el master suspenderá su ejecución durante un segundo, volviendo a comprobar si las tareas iniciales ya han acabado cuando se reanude el hilo, y así sucesivamente.
4. Cuando la ejecución de todas las tareas iniciales ha acabado, hay que elegir qué número de particiones ha sido el que ha tenido un mejor rendimiento. Para ello, se ha escogido utilizar la métrica *tiempo por token*. Puesto que en Cassandra todos los rangos de tokens tienen aproximadamente la misma cantidad de datos, esta es una buena métrica. Para calcular el *tiempo por token* de una partición se hace del siguiente modo:

Primero, se tiene que saber el número de tokens de la partición. Para ello, se hace de la misma forma que en el apartado 4. Background.

5. Ahora solo hace falta dividir el tiempo total de ejecución entre el número de tokens:

time_per_token = partition_time / tokens_per_partition

6. Para elegir la mejor granularidad, primero se descartan las tareas que hayan tenido un tiempo de ejecución menor a dos segundos, para así evitar tener demasiado overhead. La mejor granularidad será aquella que haya conseguido un tiempo por token mejor, sin embargo, si ninguna ha conseguido tener un tiempo mayor o igual a dos segundos, la mejor granularidad será la más grande.

7. A partir de aquí, todas las tareas siguientes van a tener la misma granularidad.

Experimentos

Para esta primera versión del gestor de la granularidad, se han escogido dos aplicaciones reales que se han utilizado en el equipo *Data-driven Scientific Computing*. Además, se ha creado otra aplicación que sirve para ver el potencial del proyecto en futuras aplicaciones. Se ha creado un repositorio de GitHub para ver el código de todas las aplicaciones y modelos de datos[9].

Estos experimentos se han realizado en MareNostrum 4, cuyas características ya se han explicado en el apartado 3.3. Planificación temporal – Recursos. Las bases de datos NoSQL no fueron diseñadas para ser ejecutadas en supercomputadores, cosa que dificulta el despliegue de aplicaciones en estos entornos. Los experimentos se han ejecutado en un supercomputador que cuenta con un sistema de colas, en donde los recursos se asignan dinámicamente y el usuario no conoce la asignación de estos recursos hasta que ya se esté ejecutando, por lo que la configuración de estas bases de datos es una tarea tediosa y compleja. Por ello, se han utilizado unos scripts que automatizan estas tareas de reserva de recursos, configuración de la base de datos en los diferentes nodos, y ejecución de aplicaciones [16].

Aunque MareNostrum 4 dispone de *GPFS*, un sistema de archivos compartidos en red [17] que facilita el acceso a los datos, para aprovechar todo el rendimiento que ofrece la base de datos distribuida Cassandra, se hace uso de los discos locales conectados a cada uno de los nodos donde se realiza la ejecución.

App 1 - Relaciones de IPs con fechas solapadas[10]

Esta aplicación es parte del proyecto *I-BiDaaS*[11] (Industrial-Driven Big Data as a Self-Service Solution), un proyecto financiado por la Unión Europea que apunta a empoderar tanto a expertos como a no expertos a que utilicen e interactúen fácilmente con tecnologías Big Data.

El objetivo de la aplicación es encontrar relaciones entre conexiones de usuarios. Dos usuarios estarán relacionados si se han conectado desde la misma IP en rangos de fechas solapadas. Por ejemplo, si un usuario A se ha conectado desde una IP el 17-03-2019 y el 20-03-2019, y el

[9] Repositorio de GitHub: https://github.com/adrianespejo/Dynamic-granularity-manager.
[10] Aplicación en el repositorio: https://github.com/adrianespejo/Dynamic-granularity-manager/blob/master/ip_relationships/IP_relationships_overlapping_dates.py.
[11] Página web del proyecto: http://www.ibidaas.eu/.

usuario B se ha conectado desde la misma IP el 19-03-2019, tiene que detectarse como una relación.

Consta de tres funciones que van a ejecutar el gestor de la granularidad de forma aislada, por lo tanto, se va a medir la aplicación completa y también las tres funciones por separado. Estas son:

1. **chunk_aggr**: crea las estructuras que se necesitarán para el resto de la aplicación e inserta los datos en Cassandra.
2. **get_blacklist**: cuenta los diferentes usuarios que se han conectado a través de cada IP, para así poder ver las IPs a las que se han conectado más de diez personas. Estas son consideradas IPs públicas, por lo que son descartadas.
3. **compute_IPs**: encuentra la primera y última conexión de cada usuario a cada IP y guarda las conexiones solapadas.

Se han realizado pruebas de rendimiento con diferente tamaño de datos y número de nodos, para comprobar el funcionamiento del gestor en varios escenarios:

1) ~10 millones de filas de entrada, 4 nodos de Cassandra y 9 nodos de aplicación.
2) ~20 millones de filas de entrada, 4 nodos de Cassandra y 9 nodos de aplicación.
3) ~10 millones de filas de entrada, 8 nodos de Cassandra y 17 nodos de aplicación.
4) ~20 millones de filas de entrada, 8 nodos de Cassandra y 17 nodos de aplicación.

Las métricas de rendimiento de la primera versión para esta aplicación han sido:

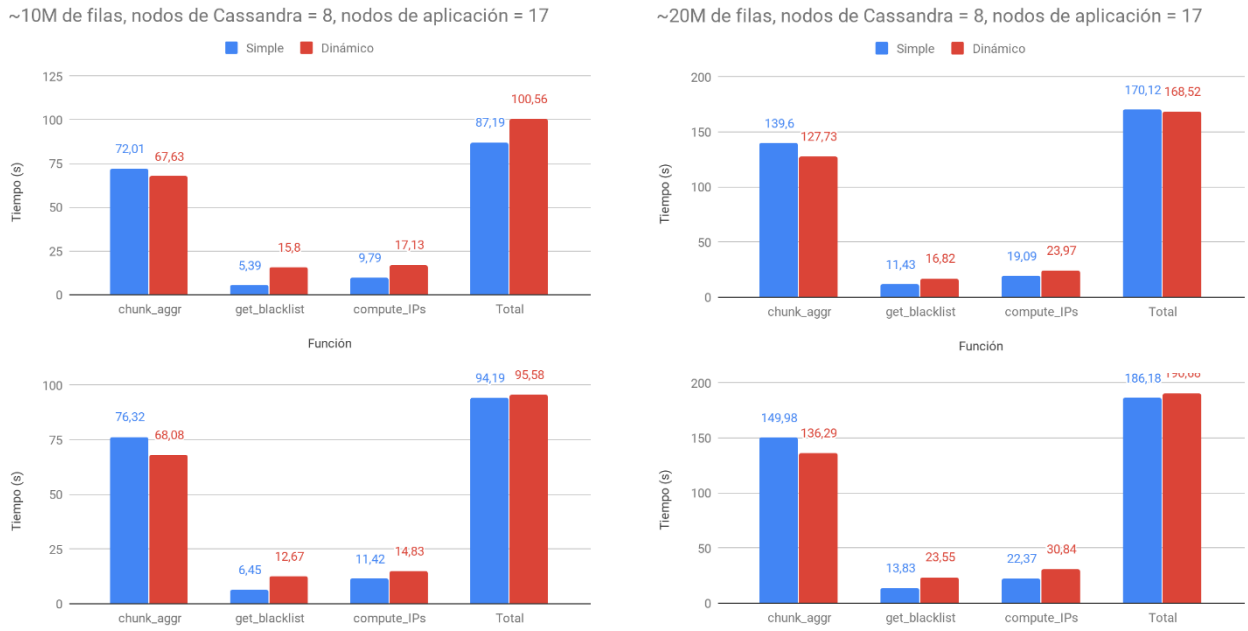


Figura 10. Rendimiento de la primera versión para la app 1

Escenario	Mejora 1a versión
(1) Filas = 10M, nC[12] = 4, nA[13] = 9	-1,48%
(2) Filas = 20M, nC = 4, nA = 9	-2,42%
(3) Filas = 10M, nC = 8, nA = 17	-15,33%
(4) Filas = 20M, nC = 8, nA = 17	0,94%

Tabla 9. Resumen de la mejora de rendimiento de la primera versión respecto a la implementación original para la app 1

Si nos fijamos en las funciones por separado de la Figura 10, vemos como hay una mejora en el tiempo de ejecución de la primera, *chunk_aggr*. Sin embargo, las otras dos funciones empeoran en todos los casos. En general los resultados no han sido satisfactorios, ya que, si hablamos de tiempo total de ejecución, esta versión ha mejorado muy poco y solo en un caso, llegando a empeorar bastante el rendimiento en el tercer escenario.

Podemos concluir que, en lo que respecta a esta aplicación, aún no merece la pena usar el gestor dinámico. Como podemos ver en la Tabla 9, en tres casos de los cuatro llega incluso a empeorar el rendimiento, por lo que habrá que intentar mejorar estos números en las siguientes versiones.

App 2 - Interpolación de métricas en diferentes longitudes y latitudes[14]

Esta es una aplicación utilizada por el departamento de *Earth Sciences* del Barcelona Supercomputing Center.

En la entrada de datos hay diferentes métricas, como por ejemplo la humedad o el ozono, que se ven identificadas por la latitud, la longitud, la elevación y el tiempo en el que fueron medidas. Para cada una de las métricas diferentes se tienen el valor mínimo, valor máximo, suma y número de mediciones. A partir de la entrada se realiza una interpolación para conseguir nuevos puntos de latitud y longitud, conservando la elevación y el tiempo.

De la misma forma que para la primera aplicación, se han realizado pruebas en varios escenarios:

1) 4MB CSV: los datos de entrada se cargan de un fichero .csv de 4MB de tamaño.

[12] nC: nodos de Cassandra.
[13] nA: nodos de aplicación.
[14] Aplicación en el repositorio: https://github.com/adrianespejo/Dynamic-granularity-manager/blob/master/earth/earth.py.

2) 51GB snapshot: la base de datos se recupera de un snapshot que se hizo en un momento determinado, y el tamaño total del snapshot es de 51GB.

Esta aplicación servirá para comprobar el rendimiento del gestor en escenarios muy desfavorables. Esto es debido a que la granularidad ideal para los dos escenarios es la granularidad que se escoge por defecto en el gestor original. Por lo tanto, los resultados que se esperan de esta aplicación no es que el gestor dinámico mejore el rendimiento, sino que el overhead de ejecución del mismo no sea demasiado grande.

Las métricas de rendimiento de la segunda versión para esta aplicación han sido:

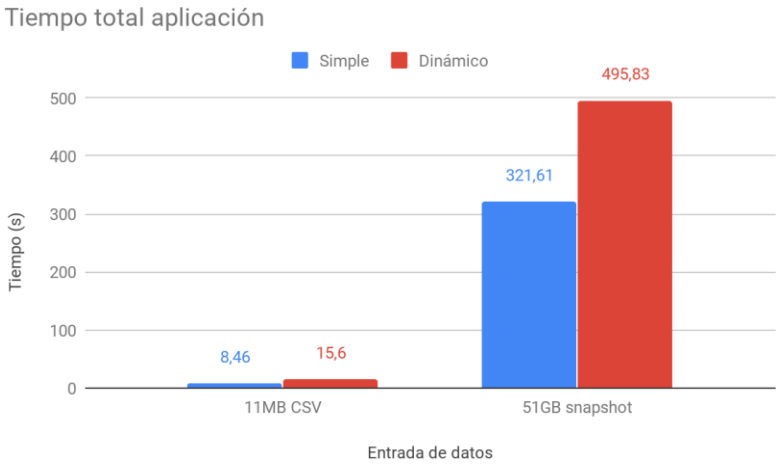


Figura 11. Rendimiento de la primera versión para la app 2

Escenario	Mejora 1a versión
(1) 4MB CSV	-84,4%
(2) 51GB snapshot	-54,17%

Tabla 10. Resumen de la mejora de rendimiento de la primera versión respecto a la implementación original para la app 2

De la Figura 11 y la Tabla 10 podemos sacar varias conclusiones:

CSV de 4MB, 4 nodos de Cassandra y 9 nodos de aplicación: la entrada de datos y la duración total de la aplicación es muy pequeña, por lo que no nos sirve para medir el speedup de la aplicación. Sin embargo, es útil para comprobar el overhead de ejecutar el gestor dinámico con aplicaciones pequeñas, el cual se puede observar que no es demasiado grande.

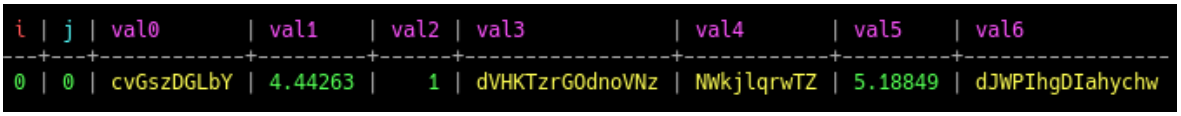
Snapshot de 51GB, 16 nodos de Cassandra y de aplicación: aquí tenemos un tamaño de entrada de datos mucho más interesante. Aun así, el gestor original es mucho más eficiente, debido a que al principio de la ejecución elige una granularidad que se acerca bastante a la óptima. El resultado es un rendimiento un 54,17% peor.

Las conclusiones de esta aplicación servirán para ver cómo funciona el gestor dinámico de la granularidad en escenarios desfavorables. Por ahora, se puede deducir que las tareas con un tamaño más grande tardan mucho, por lo que lanzar tareas más pequeñas y esperar a que acabe la más grande añade demasiado overhead.

App 3 - Aplicación Dummy[15]

Debido a la necesidad de aplicaciones donde probar el gestor de la granularidad, se ha creado una aplicación que realice un trabajo muy intensivo según los datos.

La entrada es una matriz A de tamaño N*N con diferentes atributos aleatoriamente generados por posición, por ejemplo, en la Figura 12 se puede ver los posibles atributos de la posición *[0][0]*.



i	j	val0	val1	val2	val3	val4	val5	val6
0	0	cvGszDGLbY	4.44263	1	dVHKTzrGOdnoVNz	NWkjlqrwTZ	5.18849	dJWPIhgDIahychw

Figura 12. Ejemplo de una posición de la matriz

El resultado de la aplicación es una matriz B del mismo tamaño que la inicial. Al inicio de la ejecución se calculan dos decimales de forma aleatoria, que serán *a* y *b*. Después, por cada posición de la matriz A, se escoge un número de 0 a 50 también de forma aleatoria, al que llamaremos *n*. El valor de B en la posición *[i][j]* será el resultado de sumar *a*val1 + b*val5* un total de *n* veces.

También se van a ejecutar pruebas en diferentes escenarios, estos son:

1) Tamaño de la matriz N = 1000, 4 nodos de Cassandra y 9 nodos de aplicación.
2) Tamaño de la matriz N = 1500, 4 nodos de Cassandra y 9 nodos de aplicación.
3) Tamaño de la matriz N = 1000, 8 nodos de Cassandra y 17 nodos de aplicación.
4) Tamaño de la matriz N = 1500, 8 nodos de Cassandra y 17 nodos de aplicación.

Esta aplicación es muy interesante porque el trabajo de cada tarea aumenta mucho según aumenta la granularidad, llegando a realizar hasta 100 multiplicaciones en coma flotante y también hasta 100 lecturas y escrituras por posición de la matriz.

[15] Aplicación en el repositorio: https://github.com/adrianespejo/Dynamic-granularity-manager/blob/master/dummy/Dummy.py.

Las métricas de rendimiento de la primera versión para esta aplicación han sido:

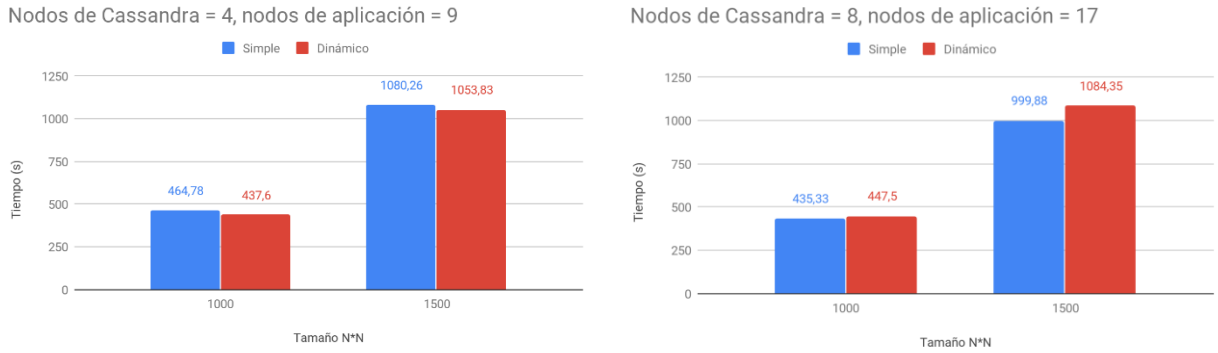


Figura 13. Rendimiento de la primera versión para la app 3

Escenario	Mejora 1a versión
(1) N = 1000, nC = 4, nA = 9	5,85%
(2) N = 1500, nC = 4, nA = 9	2,45%
(3) N = 1000, nC = 8, nA = 17	-2,8%
(4) N = 1500, nC = 8, nA = 17	-8,45%

Tabla 11. Resumen de la mejora de rendimiento de la primera versión respecto a la implementación original para la app 3

De la Figura 13 y la Tabla 11 anteriores, vemos que en dos casos tenemos una mejora del rendimiento relevante, salvo cuando se usan 8 nodos de Cassandra y 17 de aplicación, donde el rendimiento ha empeorado en ambos casos.

Los resultados son buenos para ser una primera versión, por el hecho de que en algunos casos el tiempo de ejecución es menor, aunque se espera que en experimentos posteriores se consigan resultados mucho mejores.

Conclusiones

Las métricas de rendimiento no son suficientemente buenas, pero es lo esperado dado que es la primera versión.

El mayor problema de esta versión es que, al lanzar tareas con una lista de granularidades y esperar a que acaben sin lanzar más, los workers están inactivos durante esta espera, por lo que se pierde mucho tiempo que podría aprovecharse para ejecutar tareas.

6.4. Versión 2[16]

Diseño e implementación

Esta versión tiene como objetivo mejorar la versión anterior en el aspecto de que no haya nodos inactivos durante la ejecución. Para conseguirlo, se ha añadido un contador de nodos inactivos, que es incrementado cuando se detecta que ha acabado una tarea y es decrementado cuando se lanza otra. Cuando se detecta un nuevo tiempo de ejecución de una tarea en la base de datos que no había sido capturado en el master anteriormente, se incrementa el contador de nodos inactivos.

En el momento de intentar crear una nueva partición, si ya se han lanzado las tareas con las granularidades iniciales, en la versión anterior se esperaba hasta que acabaran todas. Ahora, durante la espera se comprueba si hay algún nodo inactivo, y si es el caso se lanza una tarea con la granularidad que haya conseguido un tiempo por token mejor hasta ese momento.

Además, a diferencia de la versión anterior, para cada granularidad puede haber más de una métrica de tiempo diferente, por lo que cuando llegue el momento de calcular el mejor tiempo por token se hará una media de estas métricas.

Experimentos

Para esta versión se han utilizado las mismas aplicaciones que para la anterior, sin realizar ningún tipo de cambio.

Las métricas de rendimiento de la segunda versión para esta aplicación han sido:

[16] Código de la segunda versión: https://github.com/adrianespejo/Dynamic-granularity-manager/blob/master/partitioners/partitioner2.py.

App 1 - Relaciones de IPs con fechas solapadas

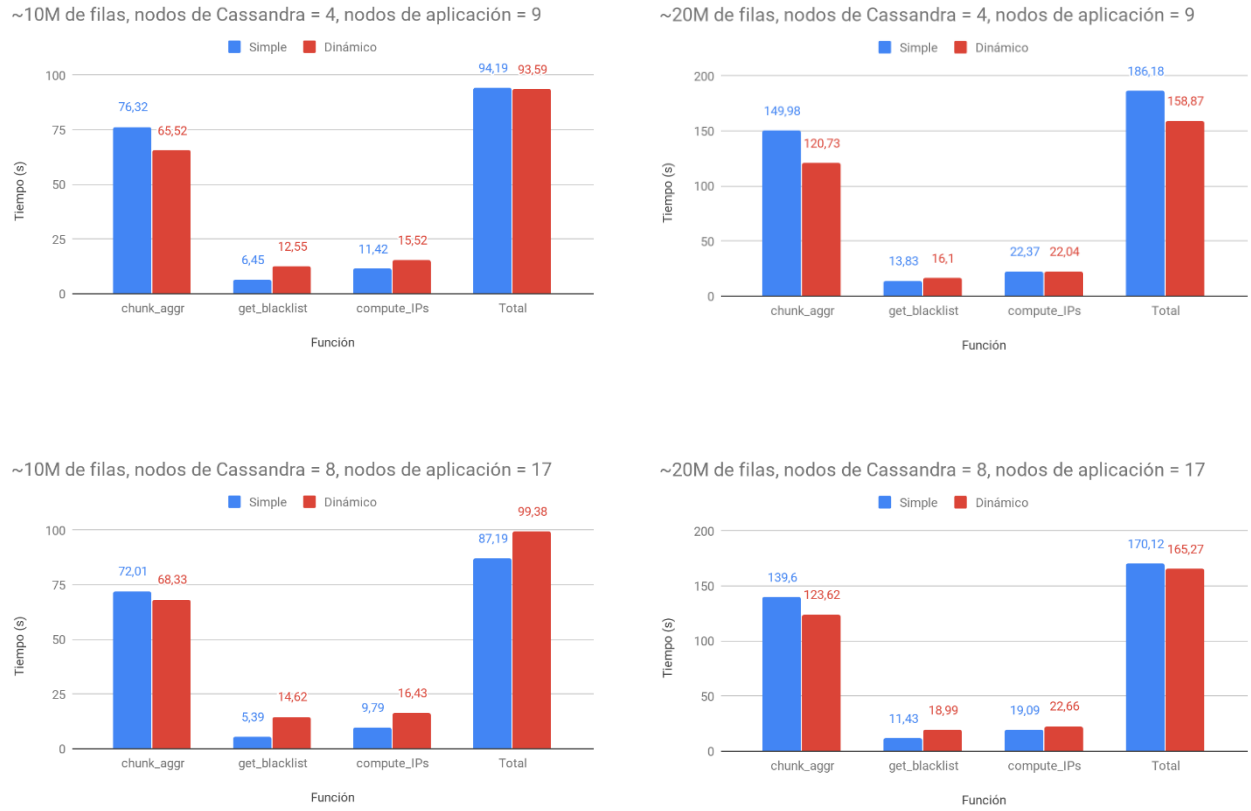


Figura 14. Rendimiento de la segunda versión para la app 1

	Mejora de X versión	
Escenario	1a	2a
1) Filas = 10M, nC = 4, nA = 9	-1,48%	0,64%
2) Filas = 20M, nC = 4, nA = 9	-2,42%	14,67%
3) Filas = 10M, nC = 8, nA = 17	-15,33%	-13,98%
4) Filas = 20M, nC = 8, nA = 17	0,94%	2,85%

Tabla 12. Resumen de la mejora de rendimiento de las dos primeras versiones respecto a la implementación original para la app 1

El rendimiento de la aplicación ha mejorado en general, aunque se puede apreciar en mayor medida en la Tabla 12 en el segundo escenario (~20M de filas, 4 nodos de Cassandra y 9 nodos de aplicación). La mejora es muy leve para los demás casos, pero aun así hay que tenerlo en cuenta.

De la misma forma que con la versión anterior, la función *chunk_aggr* tiene un mayor speedup en general, pero *get_blacklist* y *compute_IPs* siguen empeorando en todos los escenarios. Al intentar entender la razón de que las dos últimas funciones empeoren, se ha visto que la duración de las tareas es bastante pequeña, por lo que apenas se aprovecha el gestor dinámico de la granularidad.

App 2 - Interpolación de métricas en diferentes longitudes y latitudes

Las métricas de rendimiento de la segunda versión para esta aplicación han sido:

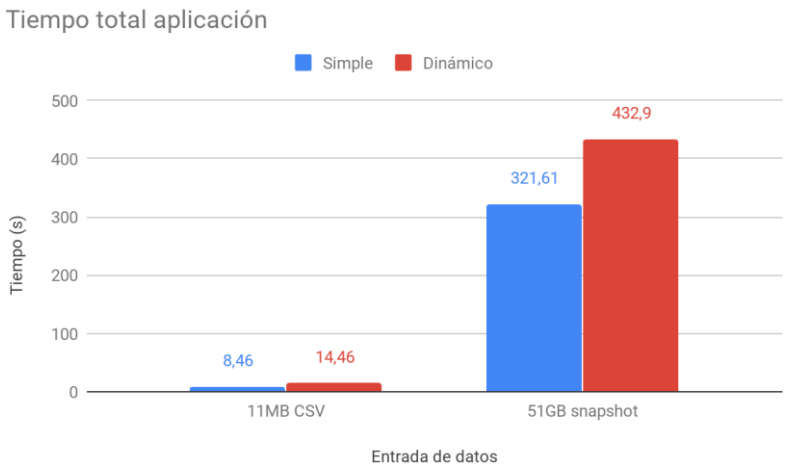


Figura 15. Rendimiento de la segunda versión para la app 2

	Mejora de X versión	
Escenario	1a	2a
1) 4MB CSV	-84,4%	-70,92%
2) 51GB snapshot	-54,17%	-34,60%

Tabla 13. Resumen de la mejora de rendimiento de las dos primeras versiones respecto a la implementación original para la app 2

Como se ha dicho antes, las conclusiones de esta aplicación sirven para ver cómo funciona el gestor dinámico de la granularidad en escenarios desfavorables.

En ambos casos el overhead se ha reducido en gran medida. Aunque en el primero el rendimiento empeora mucho comparado con el original, solo hay unos pocos segundos de diferencia. De la Tabla 13 vemos que en el segundo caso la mejora ha aumentado un 19,66%, con lo que podemos concluir que al mandar nuevas tareas siempre que algún nodo esté disponible ha reducido en gran medida el overhead.

App 3 - Aplicación Dummy

Las métricas de rendimiento de la segunda versión para esta aplicación han sido:

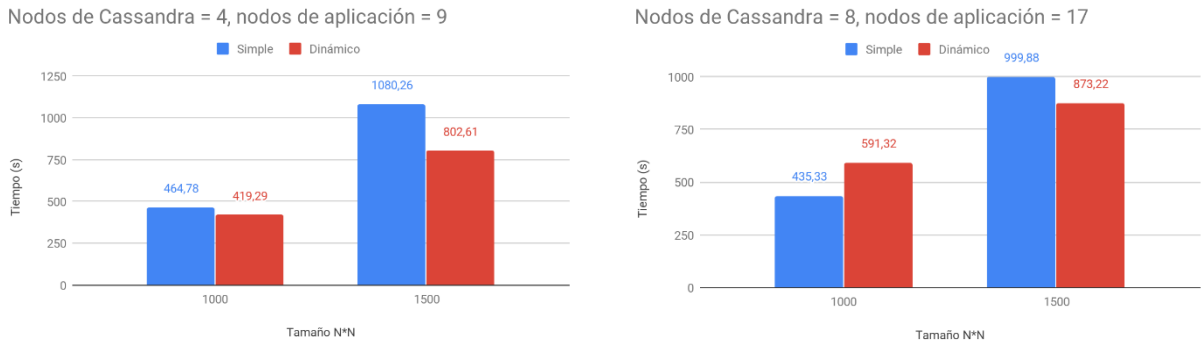


Figura 16. Rendimiento de la segunda versión para la app 3

	Mejora de X versión	
Escenario	1a	2a
1) N = 1000, nC = 4, nA = 9	5,85%	9,79%
2) N = 1500, nC = 4, nA = 9	2,45%	25,70%
3) N = 1000, nC = 8, nA = 17	-2,8%	-35,83%
4) N = 1500, nC = 8, nA = 17	-8,45%	12,67%

Tabla 14. Resumen de la mejora de rendimiento de las dos primeras versiones respecto a la implementación original para la app 3

Viendo la Tabla 14, en el segundo y cuarto escenario hay un aumento de un 23,25% y un 21,12% respecto de la versión anterior, resultados muy satisfactorios. La mejora del primer

escenario es menor, del 3,94%, aunque también es relevante. Sin embargo, en el tercero el rendimiento ha sido mucho peor, llegando a empeorar un 35,85% respecto de la versión original. Este aumento se debe a que la tarea con una granularidad más grande tarda mucho en acabar, y el hecho de lanzar nuevas tareas solamente cuando se detecte un nodo inactivo genera bastante overhead en este caso.

Conclusiones

En general, las métricas de rendimiento son ligeramente mejores que las de la versión anterior. Hay algunos casos especiales en que ha mejorado mucho el rendimiento, como los segundos escenarios de las aplicaciones Dummy y relaciones de IPs. El overhead de la aplicación de interpolación de métricas también se ha visto bastante reducido. Como caso aislado está el tercer escenario de la aplicación Dummy, que ha empeorado hasta el -35,85%.

La mejora se debe al hecho de lanzar una tarea cuando se detecte que haya un nodo inactivo. Aun así, se ha llegado a la conclusión de que tarda mucho tiempo en esperar a que acaben las tareas con una mayor granularidad, y este tiempo que se tarda en detectar que un nodo está inactivo y enviar una tarea nueva ocasiona un overhead mayor que la ganancia de usar granularidades óptimas. Es importante recordar que, al acabar la ejecución, las tareas escriben su tiempo de finalización en Cassandra, y al leer de la base de datos es cuando el master se da cuenta de que un nodo está libre, lo que puede tardar unos segundos. Esto es algo que se intentará solucionar en la siguiente versión, puesto que puede mejorar el tiempo de ejecución en todos los escenarios.

6.5. Versión 3[17]

Diseño e implementación

Esta versión optimiza la espera de finalización de las tareas iniciales de la versión anterior, ya que se descubrió que la detección de nodos inactivos era bastante ineficiente. Cuando se calculan los tiempos por token para cada granularidad, se pone el tiempo actual como tiempo final de las tareas inacabadas. La tarea con un tiempo por token mejor que el de las tareas acabadas e inacabadas será la granularidad ideal. Es de consideración señalar que la granularidad de una tarea que no haya acabado no podrá ser elegida como la ideal.

Esta optimización puede mejorar bastante el rendimiento de las aplicaciones, debido a que el método de lanzar tareas cuando se vayan detectando nodos inactivos ocasiona bastante overhead. La causa de esto ha sido explicada en las conclusiones de la segunda versión (apartado 6.4. Versión 2 – Conclusiones). De esta forma, no hace falta esperar a que acaben todas las tareas iniciales, por lo que cuando se elija una granularidad ideal, se lanzarán todas las demás tareas seguidamente. La ejecución de las tareas será más eficiente, debido a que al

[17] Código de la tercera versión: https://github.com/adrianespejo/Dynamic-granularity-manager/blob/master/partitioners/partitioner3.py.

momento en el que un worker acaba una tarea, empezará la siguiente, sin que el master lo tenga que detectar.

Experimentos

Para esta versión se han utilizado las mismas aplicaciones que para la anterior, sin realizar ningún tipo de cambio.

App 1 - Relaciones de IPs con fechas solapadas

Las métricas de rendimiento de la tercera versión para esta aplicación han sido:

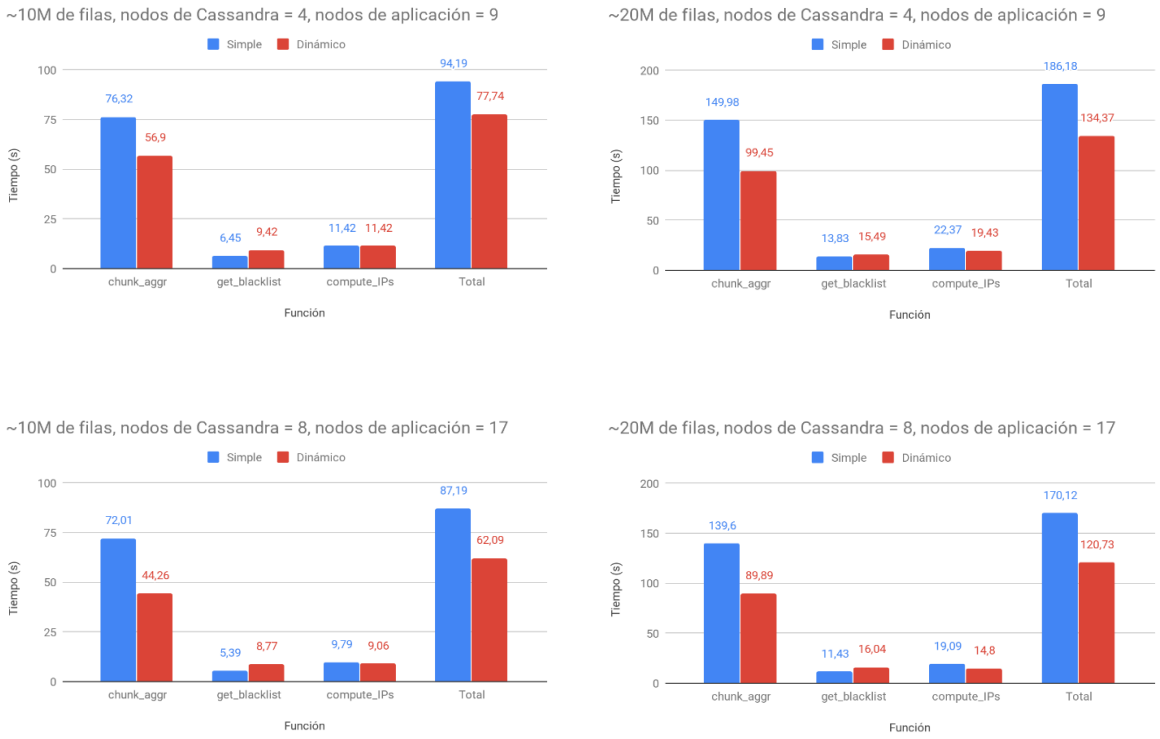


Figura 17. Rendimiento de la tercera versión para la app 1

	Mejora de X versión		
Escenario	1a	2a	3a
1) Filas = 10M, nC = 4, nA = 9	-1,48%	0,64%	17,46%
2) Filas = 20M, nC = 4, nA = 9	-2,42%	14,67%	27,83%
3) Filas = 10M, nC = 8, nA = 17	-15,33%	-13,98%	28,79%
4) Filas = 20M, nC = 8, nA = 17	0,94%	2,85%	29,03%

Tabla 15. Resumen de la mejora de rendimiento de las tres primeras versiones respecto a la implementación original para la app 1

El rendimiento en general ha mejorado drásticamente. Según la Tabla 15, de media la mejora respecto a la versión original ha sido del 25,78%, lo cual es un resultado muy satisfactorio. Esta mejora ha sido mayoritariamente debido a la función *chunk_aggr*, ya que como se ha mencionado anteriormente, las tareas que realizan las funciones *get_blacklist* y *compute_IPs* no tardan lo suficiente como para que merezca la pena usar el gestor dinámico de la granularidad.

App 2 - Interpolación de métricas en diferentes longitudes y latitudes

Las métricas de rendimiento de la tercera versión para esta aplicación han sido:

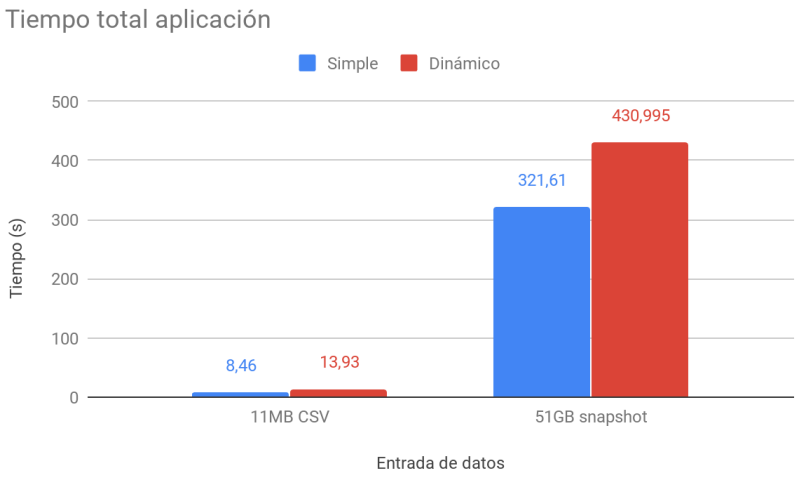


Figura 18. Rendimiento de la tercera versión para la app 2

	Mejora de X versión		
Escenario	1a	2a	3a
1) 4MB CSV	-84,4%	-70,92%	-64,66%
2) 51GB snapshot	-54,17%	-34,60%	-34,01%

Tabla 16. Resumen de la mejora de rendimiento de las tres primeras versiones respecto a la implementación original para la app 2

Observando la Tabla 16 vemos como solamente ha mejorado el rendimiento en el primer escenario, se ha reducido un poco el overhead de la ejecución del gestor dinámico. El rendimiento en el segundo es prácticamente el mismo, por lo podemos decir que no ha mejorado en lo que respecta a este escenario.

App 3 - Aplicación Dummy

Las métricas de rendimiento de la tercera versión para esta aplicación han sido:

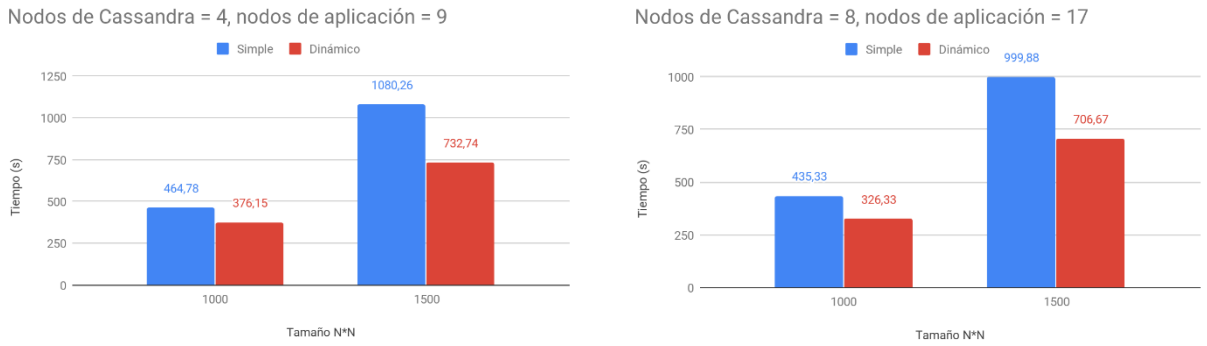


Figura 19. Rendimiento de la tercera versión para la app 3

	Mejora de X versión		
Escenario	1a	2a	3a
1) N = 1000, nC = 4, nA = 9	5,85%	9,79%	19,07%
2) N = 1500, nC = 4, nA = 9	2,45%	25,70%	32,17%
3) N = 1000, nC = 8, nA = 17	-2,80%	-35,83%	25,04%
4) N = 1500, nC = 8, nA = 17	-8,45%	12,67%	29,32%

Tabla 17. Resumen de la mejora de rendimiento de las tres primeras versiones respecto a la implementación original para la app 3

Como podemos ver en la Figura 19, hay una gran mejora en el segundo y tercer escenarios, cuando se usan 8 nodos de Cassandra y 17 nodos de aplicación. Igual que para la primera aplicación, estos resultados son muy satisfactorios ya que, según la tabla 17, se consigue una mejora del rendimiento bastante alta respecto a la implementación original, siendo del 26,4% de media.

Conclusiones

La primera y tercera aplicación mejoran, de media, un 25,78% y un 26,4%, respectivamente. Sin embargo, en lo que respecta a la segunda aplicación, hay una mejora media negativa, de un -49,33%. Como se ha dicho anteriormente, el gestor dinámico contaba con una gran desventaja en esta aplicación, por lo que los resultados no son tan malos.

Estos resultados son suficientemente buenos como para que esta sea una solución al problema, aunque el gestor aún no es totalmente dinámico, hecho que se conseguirá en la siguiente versión.

6.6. Versión 4[18]

Diseño e implementación

Última versión del proyecto, el gestor de la granularidad totalmente dinámico. Igual que las versiones anteriores, al inicio de la ejecución lanza tantas tareas como nodos disponibles, pero ahora no espera a que acaben para elegir la granularidad ideal, si no que cada vez que hay un nodo inactivo envía la tarea que haya tenido un mejor tiempo por token por el momento.

[18] Código de la cuarta versión: https://github.com/adrianespejo/Dynamic-granularity-manager/blob/master/partitioners/partitioner4.py.

Experimentos

Para esta versión se han utilizado las mismas aplicaciones que para la anterior, sin realizar ningún tipo de cambio.

App 1 - Relaciones de IPs con fechas solapadas

Las métricas de rendimiento de la cuarta versión para esta aplicación han sido:

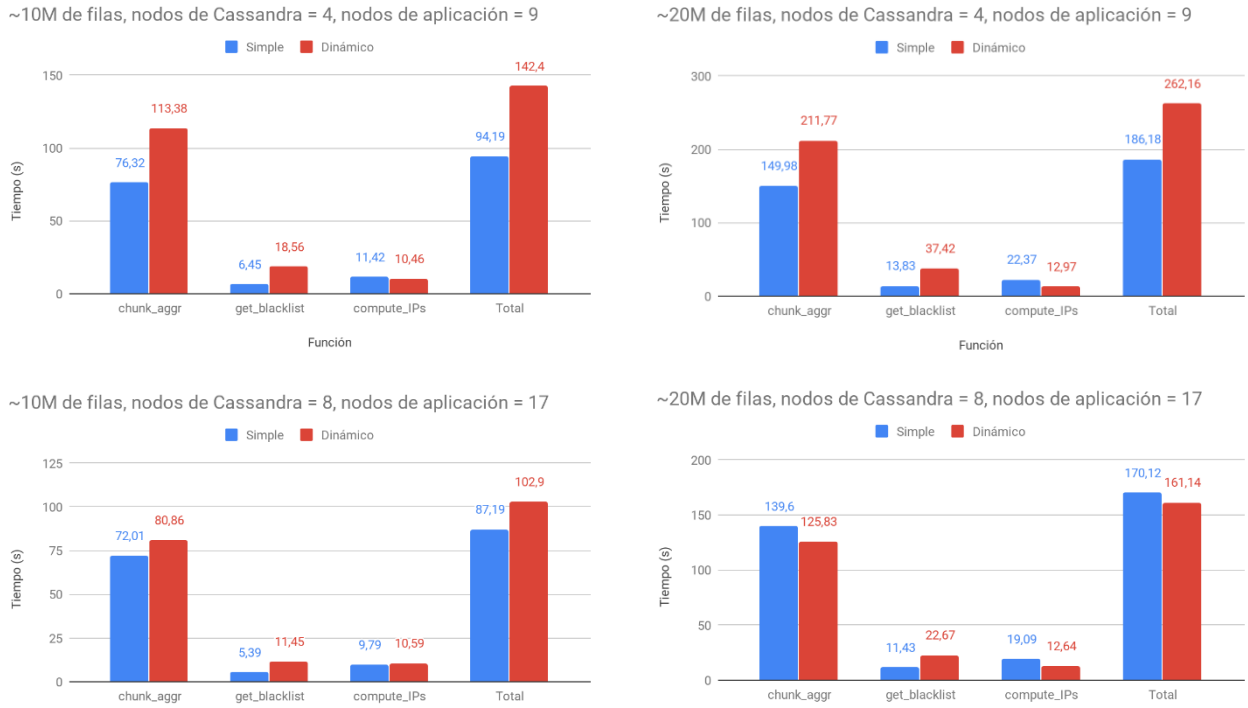


Figura 20. Rendimiento de la cuarta versión para la app 1

	Mejora de X versión			
Escenario	1a	2a	3a	4a
1) Filas = 10M, nC = 4, nA = 9	-1,48%	0,64%	17,46%	-51,18%
2) Filas = 20M, nC = 4, nA = 9	-2,42%	14,67%	27,83%	-40,81%
3) Filas = 10M, nC = 8, nA = 17	-15,33%	-13,98%	28,79%	-18,02%
4) Filas = 20M, nC = 8, nA = 17	0,94%	2,85%	29,03%	5,28%

Tabla 18. Resumen de la mejora de rendimiento de las cuatro primeras versiones respecto a la implementación original para la app 1

Como podemos ver en la Figura 20, en tres de los cuatro escenarios, el rendimiento ha empeorado en gran medida respecto a la versión original. Esto es debido a la forma que tienen de comunicar la finalización de las tareas entre workers y master, en el que puede haber unos segundos de retraso entre la finalización y el comienzo de una tarea. No obstante, en el último escenario sí que hay una ligera mejora respecto al gestor original.

App 2 - Interpolación de métricas en diferentes longitudes y latitudes

Las métricas de rendimiento de la cuarta versión para esta aplicación han sido:

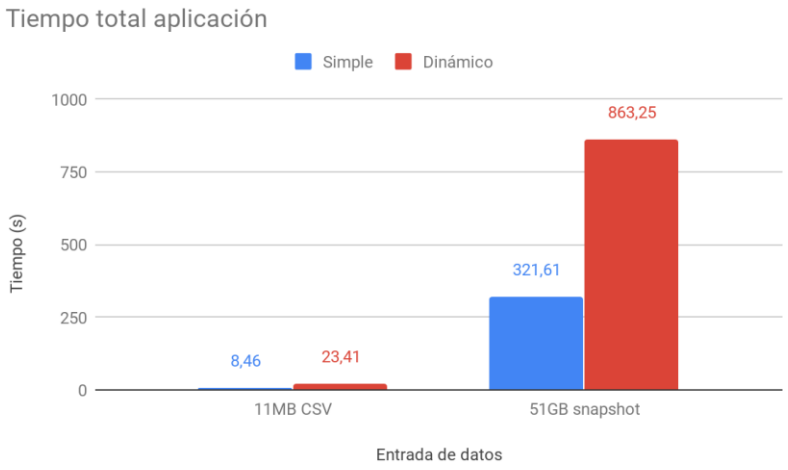


Figura 21. Rendimiento de la cuarta versión para la app 2

	Mejora de X versión			
Escenario	1a	2a	3a	4a
1) 4MB CSV	-84,4%	-70,92%	-64,66%	-176,71%
2) 51GB snapshot	-54,17%	-34,60%	-34,01%	-168,42%

Tabla 19. Resumen de la mejora de rendimiento de las cuatro primeras versiones respecto a la implementación original para la app 2

En los dos casos el rendimiento ha empeorado en gran medida. El gestor dinámico ya contaba con desventaja, y sumado al overhead mencionado en las métricas de la aplicación anterior, hace que los resultados sean muy malos.

App 3 - Aplicación Dummy

Las métricas de rendimiento de la cuarta versión para esta aplicación han sido:

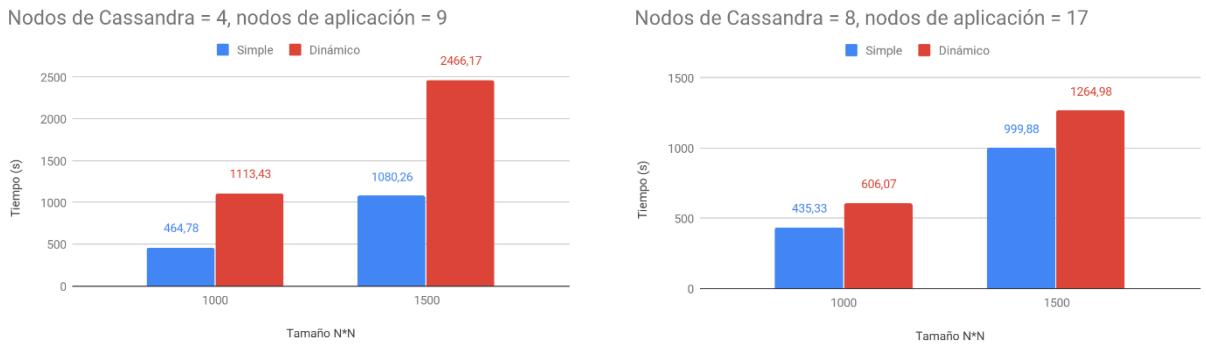


Figura 22. Rendimiento de la cuarta versión para la app 3

	Mejora de X versión			
Escenario	1a	2a	3a	4a
1) N = 1000, nC = 4, nA = 9	5,85%	9,79%	19,07%	-139,56%
2) N = 1500, nC = 4, nA = 9	2,45%	25,70%	32,17%	-128,29%
3) N = 1000, nC = 8, nA = 17	-2,80%	-35,83%	25,04%	-39,22%
4) N = 1500, nC = 8, nA = 17	-8,45%	12,67%	29,32%	-26,51%

Tabla 20. Resumen de la mejora de rendimiento de las cuatro primeras versiones respecto a la implementación original para la app 3

Igual que para las demás aplicaciones, esta versión empeora mucho el rendimiento. Sin embargo, podemos observar cómo, cuándo aumentan los datos o el número de nodos, el rendimiento no empeora tanto. Este hecho se debe a que el beneficio de lanzar tareas con granularidades óptimas equilibra en mayor medida el overhead de ejecutar el gestor dinámico cuando los datos o los nodos aumentan, aunque no lo suficiente.

6.7. Conclusiones

El rendimiento de las aplicaciones con un gestor de la granularidad totalmente dinámico ha empeorado mucho de forma general. Como se ha explicado antes, esto es debido mayormente

al overhead de las comunicaciones entre master y workers, que pueden tardar unos segundos. Cuando esto ocurre con granularidades pequeñas donde se lanzan muchas tareas, el overhead puede llegar a ser muy grande.

Gracias a las métricas de rendimiento podemos observar unos resultados que no eran los esperados. Por todo lo anteriormente dicho, esta versión del gestor no cumple con los requisitos necesarios como para ser una solución del proyecto.

6. Conclusiones del proyecto

Para poder extraer conclusiones de las pruebas de rendimiento, se han hecho varias gráficas que comparan las diferentes versiones para cada aplicación:

App 1 - Relaciones de IPs con fechas solapadas

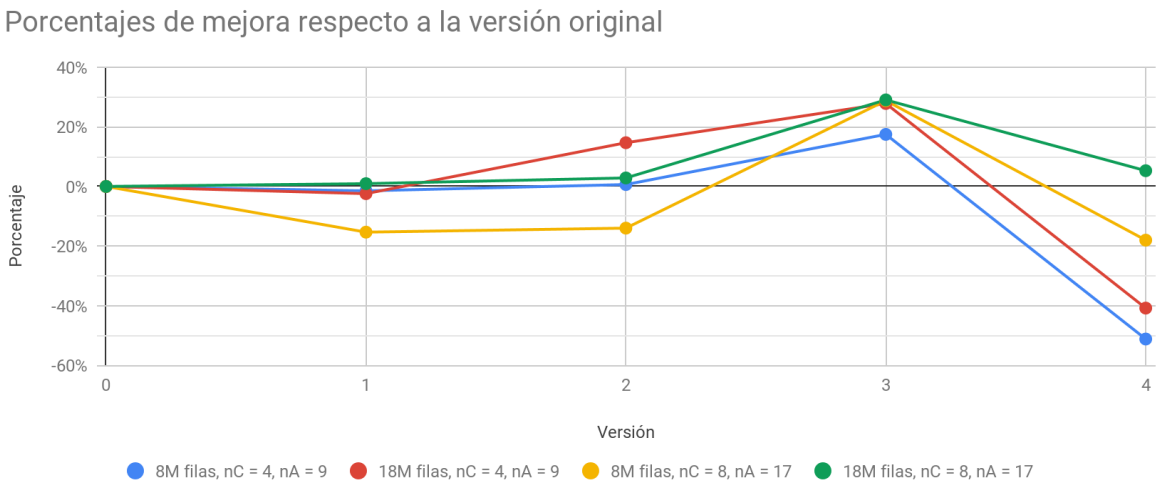


Figura 23. Comparación del rendimiento de las diferentes versiones para la app 1

Como se muestra en la Figura 23, en todos los escenarios la tercera versión del gestor dinámico de la granularidad es la mejor. La última versión, la totalmente dinámica, tiene un rendimiento peor que la tercera. Con un speedup medio del 25,78%, la mejor solución al problema es la tercera versión.

App 2 - Interpolación de métricas en diferentes longitudes y latitudes

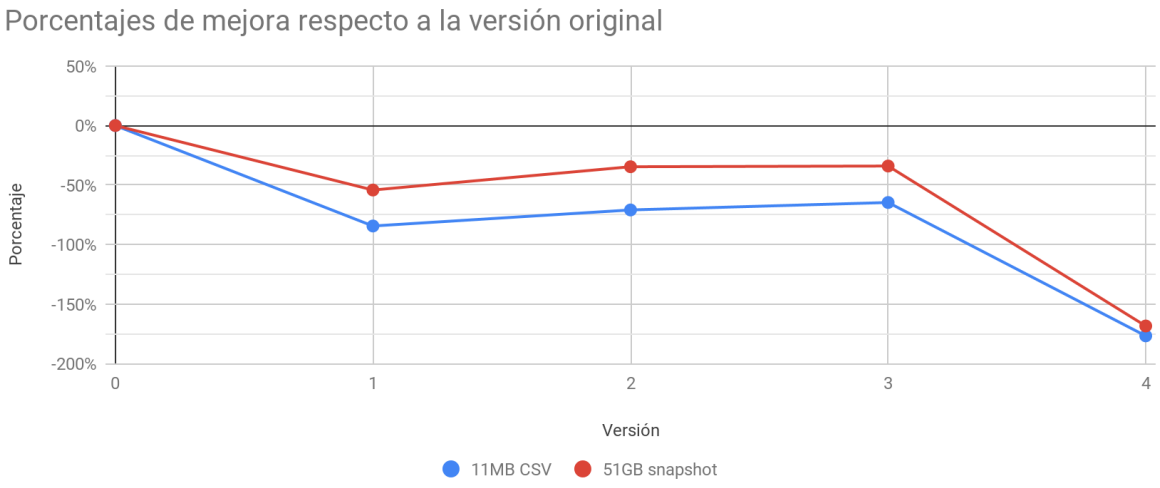


Figura 24. Comparación del rendimiento de las diferentes versiones para la app 2

Viendo la Figura 24, igual que para la aplicación anterior, la cuarta versión es peor que la tercera, aunque si nos fijamos en el eje vertical, esta vez con mucha más diferencia. La diferencia es que esta vez, la segunda y tercera versión nos ofrecen prácticamente el mismo rendimiento, por lo que cualquiera de las dos podría ser una solución al problema.

App 3 - Aplicación Dummy

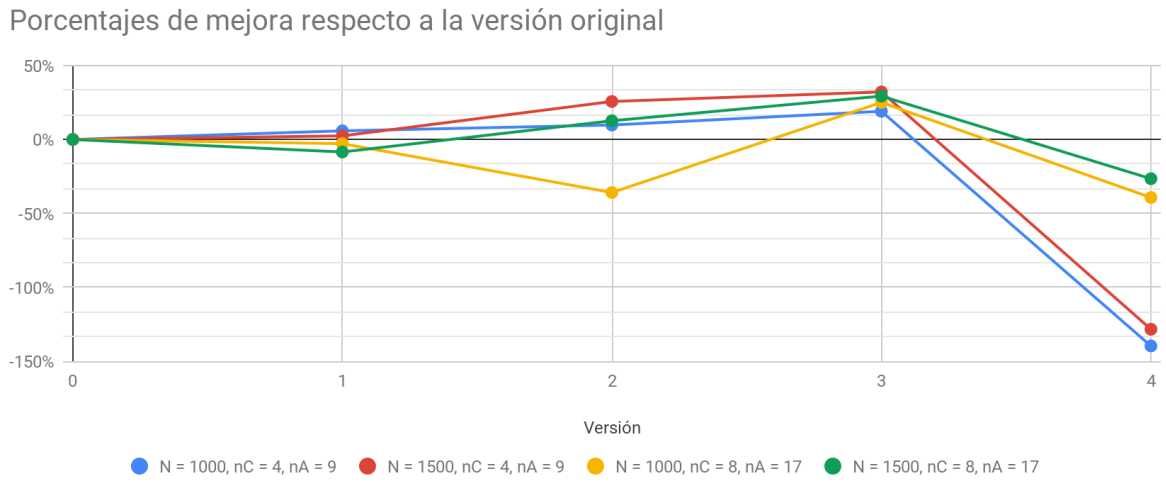


Figura 25. Comparación del rendimiento de las diferentes versiones para la app 3

Como muestra la Figura 25, de la misma forma que para la primera aplicación, la mejor versión es la tercera. La cuarta versión llega a empeorar demasiado el rendimiento en los casos con 4 nodos de Cassandra y 9 nodos de aplicación. Por ello, la mejor solución para el problema es la tercera versión, con un 26,40% de speedup de media.

Conclusión general

En general, la tercera versión es la que ofrece un mejor rendimiento, teniendo mejores métricas en dos de las tres aplicaciones probadas. Hay una aplicación en la que el rendimiento empeora, porque como se ha mencionado antes, la granularidad ideal es la que elige por defecto el gestor original, por lo que los resultados han sido los esperados.

De todas formas, la solución tiene como posibilidad utilizar el gestor de la granularidad simple (la versión original de Hecuba), el particionador por tamaño de los datos y el gestor dinámico de la granularidad. Por ello, se puede utilizar el gestor dinámico de la granularidad para ver cuál es la granularidad ideal de una aplicación, y después utilizar el gestor simple para ejecutarla eligiendo la granularidad ideal como granularidad por defecto. Cada gestor de la granularidad ofrece diferentes ventajas, por lo que los tres se pueden utilizar de forma conjunta.

El gestor de la granularidad original es el que ofrece un mejor rendimiento al no tener overhead, sin embargo, requiere de unos conocimientos y un esfuerzo mayor que los otros dos. Esto es debido a que el usuario tiene que elegir la granularidad deseada, y para ello necesita conocimientos de computación distribuida.

El gestor estático ofrece la ventaja de que la granularidad de las tareas varía según la entrada de datos que tengan, adaptándose a cada conjunto de datos. No obstante, también requiere de ciertos conocimientos de computación distribuida, ya que el usuario tiene que elegir el tamaño de entrada de cada tarea, en kilobytes.

El gestor dinámico ofrece la mayor simplicidad, con un coste de rendimiento en tiempo de ejecución. Este overhead en la mayoría de casos es pequeño, por lo que el coste de rendimiento no es muy grande. Sin embargo, la mejora del rendimiento en tiempo de programación sí que puede ser muy grande, ya que se adapta al tamaño de la entrada de los datos y al tipo de tarea que se quiera ejecutar.

La tercera versión es la mejor solución al problema que se intenta resolver, por lo que es la versión que se unificará con la rama principal de Hecuba.

Trabajo futuro

Aunque las conclusiones del proyecto han sido satisfactorias, aún hay margen de mejora. Un aspecto que se debería de mejorar en el futuro es el overhead que se añade cuando la granularidad ideal tiene un tamaño grande, ya que en algunos casos el gestor de la granularidad original tiene un rendimiento mejor que el dinámico.

En este proyecto, para saber si un nodo estaba inactivo se hace hacer una lectura de la base de datos. Si la API de PyCOMPSs para Hecuba ofreciera otros métodos más eficientes para conocer el estado de la ejecución, el rendimiento podría mejorar, sobre todo en los casos en donde se añade más overhead. Si esto no fuera posible, se deberían analizar más opciones, dado que esto es bastante ineficiente.

7. Resolución del problema

En el apartado 3.1. Alcance – Requisitos de los objetivos se plantearon varios requisitos para que los objetivos sean alcanzados de forma satisfactoria al final del proyecto. Estos son:

1. El software implementado mejora el rendimiento de la versión inicial de Hecuba.

El cumplimiento de este requisito se ha ido mencionando a lo largo de todo el apartado 5.3. Gestor dinámico y el 6. Conclusiones del proyecto gracias a las pruebas de rendimiento, por lo que no es necesario entrar mucho en detalle.

La solución escogida del gestor dinámico para el proyecto mejora el rendimiento en dos de las tres aplicaciones, y en la restante no hay demasiado overhead de ejecución. Por lo tanto, este requisito se ha cumplido satisfactoriamente.

2. Los usuarios no notarán que hay un gestor de la granularidad de las tareas detrás, solamente tendrán que elegir la estrategia del gestor.

Es necesario que el usuario añada una variable de entorno llamada PYCOMPSS_NODES con todos los nodos disponibles, para así saber la cantidad de nodos de PyCOMPSs. Esta variable de entorno ya era necesaria antes de realizar el proyecto, así que no se ha añadido ninguna entrada respecto a la implementación original. Por lo tanto, los usuarios no tienen que realizar ningún cambio a sus aplicaciones con tal de ejecutar el gestor dinámico de la granularidad, lo que significa que este objetivo se ha cumplido.

3. El gestor dinámico aproxima rápidamente la granularidad ideal, de forma que prácticamente no se pierde tiempo realizando tareas con granularidades que empeoren el tiempo de ejecución total. Además, no se penaliza demasiado el tiempo de ejecución al estar corriendo continuamente.

La granularidad ideal se elegirá en el momento en el que acabe una tarea con dicha granularidad ideal, por lo que tardará en aproximarla lo que tarde en ejecutarse. Aun así, se irán ejecutando tareas con una granularidad cada vez más próxima a la ideal, por lo que esto no es un problema. En el caso de que la granularidad ideal sea la que se elige por defecto en el gestor original, sí que hay un cierto overhead, aunque no demasiado grande. No obstante, como se ha dicho antes, se puede utilizar el gestor dinámico para averiguar cuál es la granularidad ideal, para así ejecutar la aplicación con la granularidad ideal por defecto en todas las tareas. El último objetivo también se ha cumplido.

Idealmente, la solución habría sido la versión totalmente dinámica, pero no ha sido la versión que ofrecía un mejor rendimiento. Aun así, por todo lo mencionado, el problema planteado inicialmente se ha resuelto satisfactoriamente.

8. Alcance, metodología y planificación final

8.1. Alcance final

En el capítulo 3.1. Alcance, se definieron las tres fases que tendría el proyecto, junto a sus objetivos. Estas fases eran:

1. Análisis de las opciones y diseño de la implementación.
2. Implementación y testeo del software.
3. Pruebas de rendimiento.

Cada una de las fases se ha cumplido según lo planificado, aunque ha habido algunos cambios en la planificación temporal, de los que se hablará más adelante. Como se mencionó en el mismo capítulo, las dos últimas fases se han realizado de forma iterativa, implementando una versión del software seguido de pruebas de rendimiento. Esto también se ha cumplido, realizando nuevas iteraciones hasta alcanzar una solución satisfactoria.

En dicho capítulo también se mencionaron diferentes objetivos junto a sus requisitos, y diferentes obstáculos que podían surgir durante la realización del proyecto. La explicación del cumplimiento de los requisitos se concreta en el apartado 7. Resolución del problemaResolución del problema. En lo referente a los obstáculos, solo se produjo la sobrecarga en el sistema de pruebas, pero no en suficiente medida como para variar la planificación. Al contrario, ni la ejecución del gestor penalizó demasiado el tiempo total, ni hubo problemas durante la implementación que conllevaran cambios en el diseño del software.

8.2. Metodología y rigor durante el proyecto

Durante el proyecto se han seguido principalmente dos principios de metodologías ágiles:

1. Iteraciones cortas.
2. Feedback continuo y cara a cara.

Partiendo de estos dos principios, se va a justificar su cumplimiento según las tres partes del proyecto anteriormente definidas. Estas son:

1. Análisis de las opciones y diseño de la implementación.
2. Implementación y testeo del software.
3. Pruebas de rendimiento.

Durante la primera fase estuvo muy marcado el segundo principio, porque para elegir el mejor diseño se tuvo muy en cuenta la opinión de la directora del proyecto y del equipo de

PyCOMPSs. Sin embargo, solo se realizó una iteración, debido a que aún no se había implementado nada de software.

La segunda parte se puede dividir en dos: la implementación del gestor estático por tamaño de los datos y la del gestor dinámico de la granularidad. La primera fue rápida, por lo que solo fue necesaria una iteración. No obstante, la implementación del gestor dinámico fue mucho más larga por el hecho de ser la parte principal del proyecto, además de por estar unida a las pruebas de rendimiento. Por esto, se realizaron diferentes iteraciones, que estaban compuestas por una fase de desarrollo de una nueva versión seguido de las pruebas de rendimiento.

Siempre que se implementaba una nueva versión del gestor dinámico, era una iteración diferente. Al acabar una versión había una reunión con la directora del proyecto, donde se explicaba el software implementado y su rendimiento en el caso de la fase de implementación del gestor dinámico. En estas reuniones se entendían las métricas de rendimiento que se habían tomado, para así plantear las diferentes posibilidades que podrían mejorar el rendimiento. Por ello, estos dos principios se han cumplido durante toda la duración de las fases de implementación y testeo del software y las pruebas de rendimiento.

Por otra parte, se han creado tests unitarios[19] para comprobar el buen funcionamiento del software, aunque debido a la dificultad de ejecutar dichas pruebas con PyCOMPSs de forma distribuida, estos no han sido todo lo exhaustivos como se deseaba al principio. Aun así, los tests que se han creado para su ejecución en un entorno local han cumplido la función de comprobar el correcto funcionamiento del software.

La validación de la solución es una de las partes principales del proyecto, puesto que se quería que la solución ofreciera el mejor rendimiento posible. Para ello, se ha ido tomando métricas de rendimiento de cada una de las versiones del gestor dinámico de la granularidad implementadas, y la validación de una versión venía determinada por tener el mejor rendimiento y por el cumplimiento de los objetivos mencionados en el apartado 7. Resolución del problema.

8.3. Planificación temporal final

La planificación temporal descrita en el apartado 3.3. Planificación temporal no se ha cumplido completamente por dos razones:

1) Se consiguió una primera versión del software antes de lo esperado.

2) Se necesitaba un período de tiempo específico para la escritura de la memoria.

Por la primera razón, el comienzo de la ejecución de los tests de rendimiento se adelantó, ya que era la siguiente tarea a realizar. Esto fue una muy buena decisión, debido a que ejecutar los tests en MareNostrum 4 fue una tarea muy lenta, debido al proceso de reservar la cantidad deseada de nodos, levantar la base de datos, escribir los datos necesarios, ejecutar la aplicación

[19] Tests unitarios en el repositorio: https://github.com/adrianespejo/Dynamic-granularity-manager/blob/master/partitioners/partitioner_tests.py.

y recoger los resultados. Mientras esto ocurría de forma automatizada, se podía dedicar más tiempo a la implementación del software.

Para solventar el problema de la escritura de la memoria, se asignaron dos semanas que se dedicaron a solamente esta tarea, que acabaron siendo suficientes.

Diagrama de Gantt definitivo

Primero, se ha actualizado la cantidad de horas dedicadas a cada tarea. Además, se ha incluido la escritura de la defensa como parte de la última tarea, la preparación de la memoria y la defensa.

Como se explicó en la planificación, en el caso de la gestión del proyecto se han dedicado unas dos horas al día de trabajo, mientras que para las demás se han dedicado unas cuatro horas al día. La Tabla 21 refleja las horas dedicadas a cada tarea.

Tarea	**Tiempo dedicado (h)**
Gestión del proyecto	80
Análisis de las opciones y diseño de la implementación	56
Gestor por tamaño de los datos	56
Gestor dinámico	130
Tests de rendimiento	46
Preparación de la memoria y la defensa	84
Total	452

Tabla 21. Tiempo dedicado a cada tarea

A continuación, el nuevo resumen de las tareas en un diagrama de Gantt:

Riesgo	Nombre de la tarea	Fecha de Inicio	Fecha final	Duración	Predecesoras
	Proyecto				
	Gestión del proyecto	**18/02/19**	**29/03/19**	**40d**	
	Alcance y contextualización	18/02/19	26/02/19	9d	
	Planificación temporal	27/02/19	04/03/19	6d	3
	Gestión económica y sostenibilidad	05/03/19	11/03/19	7d	4
	Documento final	12/03/19	20/03/19	9d	5
	Presentación oral	21/03/19	29/03/19	9d	6
	Análisis de las opciones y diseño de la implementación	**30/03/19**	**12/04/19**	**14d**	
	Análisis de las opciones	30/03/19	05/04/19	7d	
	Diseño de la implementación	06/04/19	12/04/19	7d	9
	Gestor por tamaño de los datos	**13/04/19**	**26/04/19**	**14d**	
	Implementación del software	13/04/19	19/04/19	7d	
	Tests unitarios	20/04/19	22/04/19	3d	12
	Tests clúster Minerva	23/04/19	26/04/19	4d	13
	Gestor dinámico	**27/04/19**	**09/06/19**	**44d**	
	Implementación del software	27/04/19	09/06/19	44d	
	Tests unitarios	11/05/19	09/06/19	30d	
	Tests clúster Minerva	14/05/19	09/06/19	27d	
	Tests de rendimiento	**18/05/19**	**09/06/19**	**23d**	
	Creación de los tests	18/05/19	21/05/19	4d	
	Realización de los tests en Marenostrum 4	22/05/19	09/06/19	19d	20
	Sacar conclusiones de los tests de rendimiento	25/05/19	09/06/19	16d	
	Preparación de la memoria y la defensa	**10/06/19**	**30/06/19**	**21d**	
	Escritura de la memoria	10/06/19	23/06/19	14d	
	Preparación de las diapositivas	24/06/19	27/06/19	4d	24
	Preparación de la defensa	28/06/19	30/06/19	3d	25

Figura 26. Tareas definitivas para el diagrama de Gantt

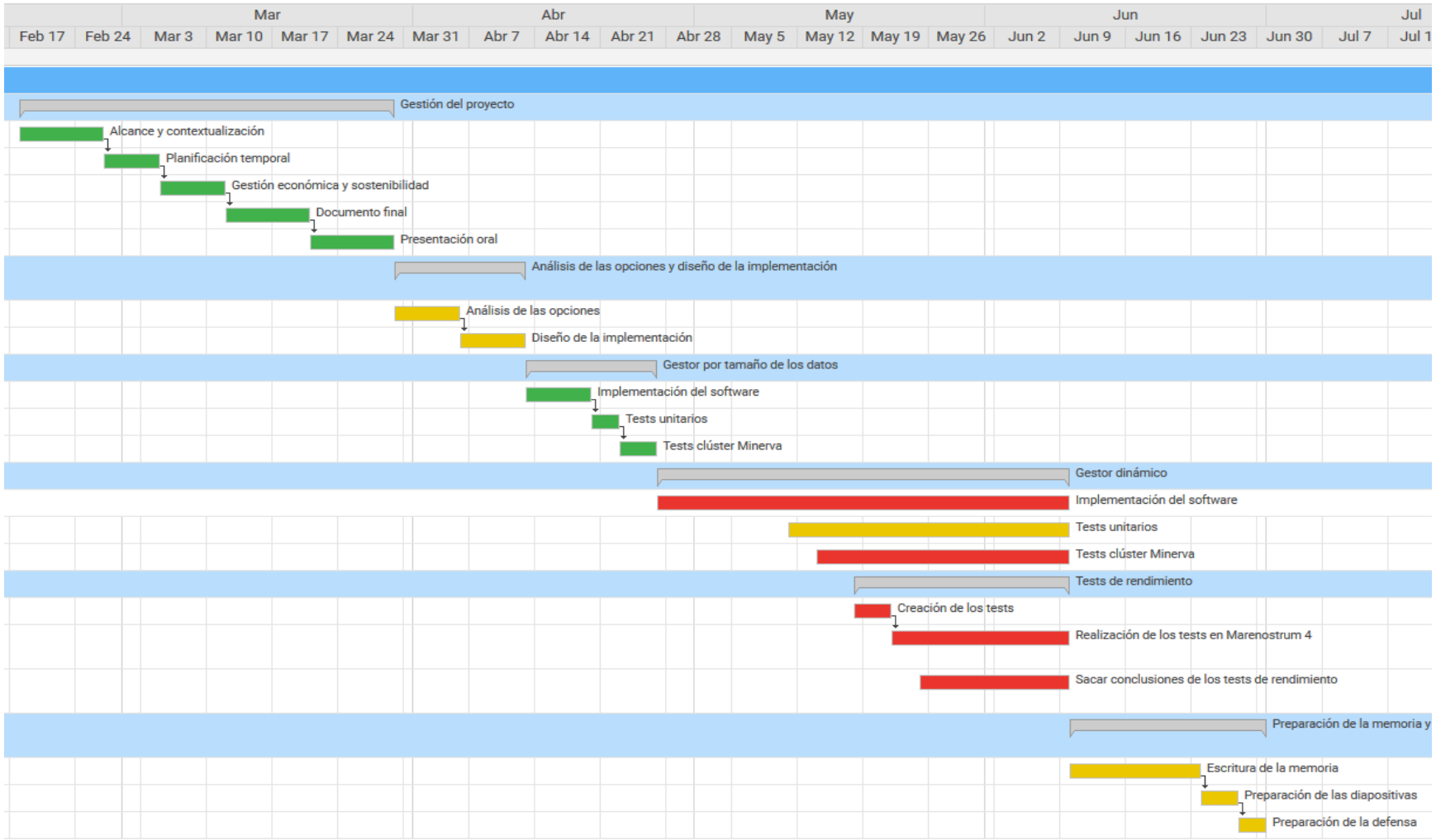


Figura 27. Diagrama de Gantt final[20]

[20] Generado con https://app.smartsheet.com/.

8.4. Presupuesto final

El presupuesto total del proyecto ha variado respecto al estimado en el apartado 3.4. Presupuesto. Esta variación se debe a que la duración de algunas tareas se ha visto alterada, por lo que el tiempo de trabajo de ciertos roles de recursos humanos también se ha visto alterado. Como el presupuesto de recursos humanos se estimó asignando un coste por hora a cada rol, el coste final es diferente al estimado.

A fin de estimar los costes totales, se hizo una división de los roles según las tareas que realizarían (ver Tabla 3). La siguiente es la tabla definitiva de horas por rol:

Tarea	**Horas (h)**	**Dedicación por rol (h)**			
		G	**A**	**D**	**T**
Gestión del proyecto	80	80			
Análisis de las opciones y diseño de la implementación	56		56		
Gestor por tamaño de los datos	56			42	14
Gestor dinámico	130			79	51
Tests de rendimiento	46				46
Preparación de la defensa	84	84			
Total	**452**	**164**	**56**	**121**	**111**

Tabla 22. Dedicación final de horas por rol

A partir de la tabla anterior y de los costes por hora de cada rol (ver Tabla 2), se ha calculado el coste final de recursos humanos:

Rol	**Horas dedicadas (h)**	**Coste final (€)**	**Coste planificado (€)**
Gestor de proyectos software	164	3.280	2.160
Analista de software	56	784	784
Desarrollador de software	121	1.573	2.028
Tester de software	111	1.110	1.320
Total	**452**	**6.747**	**6.292**

Tabla 23. Coste final de recursos humanos

Como podemos observar en la tabla anterior, debido al aumento de horas del rol gestor de proyectos software, el rol con un coste por hora mayor, el coste final de recursos humanos ha aumentado respecto al de la planificación.

Como ya se explicó en el apartado 3.4. Presupuesto – Control de presupuesto, el presupuesto de recursos humanos era el más probable que sufriera cambios. Por otra parte, los presupuestos de recursos software/hardware era muy difícil que necesitaran cambios, y efectivamente ha sido así, gracias a que los recursos que se nombraron en el apartado 3.3. Planificación temporal – Recursos han sido suficientes para la completa realización del proyecto.

Gracias a que se añadió un presupuesto para imprevistos, el aumento de los gastos se ha podido solventar. Los gastos finales han sido:

Concepto	**Presupuesto (€)**
Recursos humanos	6.747,00
Recursos de software	114,13
Recursos de hardware	217,83
Gastos generales	8.500,00
Total	**15.578,96**

Tabla 24. Coste final del proyecto

Para acabar, el coste final del proyecto contando impuestos (21%) es de **18.850,55 €**. El presupuesto del proyecto era de 19.510,00 €, por lo que los gastos han entrado dentro del presupuesto.

8.5. Sostenibilidad y compromiso social

En este capítulo se va a realizar un análisis de la sostenibilidad y compromiso social del proyecto. Para ello, se van a responder las preguntas de la matriz de sostenibilidad, a calcular el grado de sostenibilidad, y a cuantificar el impacto económico, social y ambiental. Después, se va a hacer un análisis de riesgos y a proponer formas de reducirlos y, para acabar, se presentarán las conclusiones personales sobre la sostenibilidad del proyecto.

Impacto económico

En el apartado 8.4. Presupuesto final, se ha representado el consumo de recursos (materiales y humanos) durante la realización de todo el proyecto y el coste de dichos recursos. Este presupuesto final se ha calculado teniendo en cuenta la planificación temporal detallada en el apartado 8.3. Planificación temporal final.

El coste no se ha podido reducir, debido a que el presupuesto estaba bastante acotado y era difícil que cambiara. En cambio, el coste de recursos humanos ha aumentado respecto al

presupuesto inicial, consecuencia de los cambios en la planificación temporal. Aun así, el coste final ha entrado dentro del presupuesto total inicial, gracias a haber añadido un porcentaje para imprevistos, por lo que el presupuesto se ha cumplido. Al usar metodologías ágiles ya se había previsto que el coste de recursos humanos podía verse alterado, y el añadido para imprevistos se añadió mayoritariamente por esta razón.

El coste del proyecto durante su vida útil será el mínimo: un trabajador, que hará los roles de desarrollador y tester, y un portátil, que puede ser el mismo que el del proyecto. Este trabajador se dedicará a realizar los ajustes, actualizaciones y reparaciones durante la vida útil del proyecto. Para hacer pruebas se pueden utilizar otra vez los clústeres Minerva y MareNostrum 4, que como se explicó en el capítulo 3.4. Presupuesto, su coste es nulo.

Es posible que se produzcan escenarios que perjudicasen la viabilidad del proyecto. El más obvio es que el equipo Data-driven Scientific Computing se quedase sin financiación, aunque esto es difícil que ocurra, gracias a su participación en proyectos europeos. Otro escenario es que Hecuba dejase de desarrollarse, teniendo como consecuencia que el mismo proyecto también se abandonase. Por desgracia, ninguno de estos escenarios tiene alguna medida preventiva posible.

Impacto social

La realización de este proyecto aún no me ha servido para decidir si dedicar mi carrera a la investigación. Sin embargo, sí que me ha ayudado a decidir que quiero continuar especializándome en los ámbitos del Big Data y la computación distribuida. Por otra parte, no ha implicado ninguna reflexión a nivel ético de las personas que han intervenido en el proyecto.

Respecto a la vida útil del proyecto, los usuarios de Hecuba se van a ver beneficiados del uso del software implementado. Gracias a la solución del proyecto, los usuarios de Hecuba no necesitarán tantos conocimientos de computación distribuida, y aun así verán una mejora en el rendimiento de sus aplicaciones. Al dedicar menos tiempo al desarrollo y ejecución de las aplicaciones su investigación se verá impulsada, por lo que la solución del proyecto mejora la calidad de vida de los usuarios. Por otra parte, no hay ningún colectivo que pueda verse ni directa ni indirectamente perjudicado por el proyecto.

Como se ha explicado en el apartado 7. Resolución del problemaResolución del problema, la solución del proyecto resuelve totalmente el problema planteado inicialmente, gracias al cumplimiento de los requisitos de los objetivos.

En ningún caso podría producirse un escenario en el que el proyecto fuese perjudicial para algún segmento particular de la población. Además, la intención del proyecto es la de facilitar el trabajo a científicos con pocos conocimientos de programación, por lo que sería contradictorio que algún segmento de la población pueda verse perjudicado. Sin embargo, sí que es posible que la solución del proyecto crease algún tipo de dependencia que dejase a los usuarios en posición de debilidad, y que no sepan programar de forma distribuida si no es gracias a Hecuba y PyCOMPSs. Esto no se espera que sea un problema importante, ya que, utilizando estas dos herramientas con frecuencia, se cree que los programadores ganen estos

conocimientos gracias a la experiencia obtenida con el tiempo. A pesar de esto, otra opción para reducir este riesgo es que los programadores se sigan formando en materia de computación distribuida, mientras aprovechan las ventajas que ofrece el proyecto.

Impacto ambiental

Se va a hablar del impacto ambiental a partir de los recursos utilizados, de los que se ha hablado en detalle en la planificación del proyecto, en concreto en el capítulo 3.3. Planificación temporal – Recursos.

Como las horas dedicadas han sido las estimadas en la planificación, y tampoco se ha necesitado utilizar ningún recurso adicional, el impacto ambiental del proyecto durante su realización ha sido el que ya se estimó en el apartado 3.5. Sostenibilidad – Sostenibilidad ambiental.

Debido al continuo uso del portátil, su gasto ha sido de aproximadamente 27120 W, dado que la duración del proyecto es de 452 horas y el consumo medio es de 60 Whr. El gasto es equivalente a 10441 kg de CO2. Este consumo energético ha sido algo normal y necesario.

Como ya se explicó en la sostenibilidad ambiental de la planificación, calcular el consumo de Minerva y MareNostrum 4 es muy complicado. Sin embargo, dado que su uso ha sido puntual, podemos asumir que el consumo ha sido negligible.

Para reducir el impacto ambiental, principalmente se ha reducido lo máximo posible el uso de estos clústeres. Además, gracias a que Minerva es un clúster obsoleto que se está reutilizando, el impacto ambiental ha sido menor que si se hubiera utilizado un clúster propio. Por esta razón, se cree que no habría sido posible realizar el proyecto con menos recursos si se hiciera de nuevo, porque a ello se le suma la necesidad de probar el rendimiento del software implementado en un entorno de altas prestaciones, en este caso MareNostrum 4.

Los recursos que se usarán durante la vida útil del proyecto son los mismos que se utilizaban anteriormente en las aplicaciones que usan Hecuba, por lo que la solución no necesita ningún recurso extra. No obstante, ya que el tiempo que los usuarios dedicarán a la programación y a la ejecución de aplicaciones se va a ver reducido, a su vez, el tiempo que se van a utilizar estos recursos se verá reducido. Esto implica un impacto ambiental menor de estos recursos.

Además, la solución del proyecto permitirá reducir el uso de otros recursos. Es posible que, gracias a esta mejora, otros programadores decidan incorporar Hecuba a sus aplicaciones, lo que reducirá el impacto ambiental de estas. Por lo tanto, globalmente, el uso del proyecto mejorará la huella ecológica. Esta reducción del impacto ambiental puede ser de gran cantidad, puesto que estas aplicaciones se suelen ejecutar en entornos de computación de alto rendimiento.

Con todo, existe la posibilidad de que aumente la huella ecológica del proyecto. Estos son los casos en los que la mejor granularidad sea la que se escogía por defecto en el gestor de la granularidad original, casos en los que la solución del proyecto añade cierto overhead. Estos escenarios son raros, y para mitigar este riesgo se puede comprobar el rendimiento del gestor

dinámico junto al del gestor de la granularidad original, para así ver cuál tiene mejor rendimiento y, por lo tanto, mejor huella ecológica en cada caso.

Conclusiones de la sostenibilidad

Para calcular el grado de sostenibilidad del proyecto, en la Tabla 25 se ha hecho la matriz de sostenibilidad.

	PPP	**Vida útil**	**Riesgos**
Económico	8	8	6
Social	7	10	8
Ambiental	8	10	7

Tabla 25. Matriz de sostenibilidad

Como conclusiones personales acerca sobre la sostenibilidad del proyecto, pienso que este proyecto tiene un muy buen grado de sostenibilidad. En general, realizarlo no ha requerido de muchos recursos, y seguramente tampoco los necesite en el futuro. Además, como la intención del proyecto es facilitar la programación de aplicaciones a usuarios, ganando todo el speedup posible, destaca sobre todo en la sostenibilidad social y ambiental de su vida útil. Por una parte, porque el colectivo científico puede verse muy beneficiado por el proyecto, y por otra parte porque un uso más eficiente de recursos de computación de altas prestaciones puede resultar en un gasto energético mucho menor.

9. Competencias técnicas

Este proyecto tiene asociadas las siguientes competencias técnicas:

1. CES1.4: Desarrollar, mantener y evaluar servicios y aplicaciones distribuidas con soporte de red. [En profundidad]

2. CES1.7: Controlar la calidad y diseñar pruebas en la producción de software. [Bastante]

La primera competencia se ha desarrollado durante el diseño y la implementación del gestor dinámico de la granularidad, ya que su funcionamiento debía ser distribuido. Se ha definido la interacción entre master y workers, las políticas de lanzamiento de tareas, la espera de nodos inactivos... todo de forma distribuida. Además, se han programado aplicaciones que aprovechan esta computación distribuida.

La segunda competencia se ve más marcada en la parte de las pruebas de rendimiento, dado que era una parte importante del proyecto. Por otra parte, aun teniendo en cuenta la dificultad de ejecutar pruebas unitarias con PyCOMPSs de forma distribuida, dichas pruebas han sido suficientes para comprobar el correcto funcionamiento del software antes de sus correspondientes pruebas de rendimiento. Además, como se ha mencionado a lo largo del proyecto, se ha utilizado el clúster Minerva para comprobar que la ejecución distribuida de aplicaciones era correcta, antes de ejecutar con MareNostrum 4 y así agilizar la depuración de errores.

10. Bibliografía

[1] Alomar, Guillem; Becerra Fontal, Yolanda; Torres Viñals, Jordi. Hecuba: NoSql made easy. A: "BSC Doctoral Symposium (2nd: 2015: Barcelona)". 2nd ed. Barcelona: Barcelona Supercomputing Center, 2015, p. 136-137. http://hdl.handle.net/2099/16589.

[2] Lakshman, A., Prashant, F., & Facebook, M. (n.d.). Cassandra-A Decentralized Structured Storage System. https://doi.org/10.1145/1773912.1773922.

[3] Tejedor, E., Becerra, Y., Alomar, G., Queralt, A., Badia, R. M., Torres, J., … Labarta, J. S. (2017). PyCOMPSs: Parallel computational workflows in Python. Original Article The International Journal of High Performance Computing Applications, 31(1), 66–82. https://doi.org/10.1177/1094342015594678.

[4] Partitioners. (n.d.). Retrieved June 17, 2019, from https://docs.datastax.com/en/archived/cassandra/3.0/cassandra/architecture/archPartitionerAbout.html.

[5] Dean, J., & Ghemawat, S. (2008). MapReduce: Simplified data processing on large clusters. Communications of the ACM, 51(1), 107. http://doi.org/10.1145/1327452.1327492.

[6] Muthuvelu, N., Vecchiola, C., Chai, I., Chikkannan, E., & Buyya, R. (2013). Task granularity policies for deploying bag-of-task applications on global grids. Future Generation Computer Systems, 29, 170–181. https://doi.org/10.1016/j.future.2012.03.022.

[7] Beck, K., Beedle, M., Van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., ... & Kern, J. (2009). Agile manifesto, 2001. URL https://agilemanifesto.org/principles.

[8] unittest - Python 2.7.16rc1 documentation: https://docs.python.org/2/library/unittest.html.

[9] Travis CI - Test and Deploy Your Code with Confidence: https://travis-ci.org/.

[10] DataStax Docs: https://docs.datastax.com/.

[11] Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M., & Lewin, D. (1997). Consistent hashing and random trees. In Proceedings of the twenty-ninth annual ACM symposium on Theory of computing - STOC '97 (pp. 654–663). New York, New York, USA: ACM Press. https://doi.org/10.1145/258533.258660.

[12] The most important thing to know in Cassandra data modeling: The primary key | DataStax. (n.d.). Retrieved June 17, 2019, from https://www.datastax.com/dev/blog/the-most-important-thing-to-know-in-cassandra-data-modeling-the-primary-key.

[13] Consistent hashing. (n.d.). Retrieved June 15, 2019, from https://docs.datastax.com/en/archived/cassandra/3.0/cassandra/architecture/archDataDistributeHashing.html.

[14] DataStax. (n.d.). How is data written? | Apache Cassandra 3.0. Retrieved February 18, 2019, from https://docs.datastax.com/en/cassandra/3.0/cassandra/dml/dmlHowDataWritten.html.

[15] About the nodetool utility. (n.d.). Retrieved June 23, 2019, from https://docs.datastax.com/en/archived/cassandra/3.0/cassandra/tools/toolsAboutNodetool.html.

[16] Guerrero, E. G., & Fontal, Y. B. (2018). Providing key-value data services on HPC infrastructures. Retrieved from https://upcommons.upc.edu/bitstream/handle/2117/127547/134764.pdf?sequence=1&isAllowed=y.

[17] Schmuck, F., & Haskin, R. (n.d.). USENIX Association Proceedings of the FAST 2002 Conference on File and Storage Technologies THE ADVANCED COMPUTING SYSTEMS ASSOCIATION GPFS: A Shared-Disk File System for Large Computing Clusters. Retrieved from https://www.usenix.org/legacy/events/fast02/full_papers/schmuck/schmuck.pdf.

11. Anexo

Anexo 1. Recuento de palabras usando Hecuba y PyCOMPSs[21]

```
from hecuba import StorageDict

class Words(StorageDict):
    '''
    @TypeSpec dict<<position:int>, words:str>
    '''


class Result(StorageObj):
    '''
    @TypeSpec dict<<word:str>, instances:int>
    '''


from pycompss.api.task import task
from pycompss.api.api import compss_barrier
@task
def WordCountTask(partition, result):
    for words in partition.values():
        parsedWords = words.split(',')
        for word in parsedWords:
            result[word] = result.get(word, 0) + 1

if __name__ == "__main__":
    words = Words('words')
    result = Result('result')
    for partition in words.split():
        WordCountTask(partition, result)
    compss_barrier()
```

Como se ha explicado en la introducción, primero se define el modelo de datos y se instancian los objetos persistentes. Las palabras de cada línea están separadas por comas en el objeto *words*, así que se llama a la función *split()* para crear diferentes particiones de los datos y ejecutar cada una de las funciones *WordCountTask* en paralelo, dejando el resultado en *result*. Para ejecutar una función de forma distribuida basta con poner el decorador *@task* en la función para que se ejecute en paralelo.

[21] Aplicación recuento de palabras en el repositorio de GitHub: https://github.com/adrianespejo/Dynamic-granularity-manager/tree/master/WordCount.

Anexo 2. Función para la creación de particiones.

```
def _tokens_partitions(self, tokens, min_number_of_tokens):
    """
    Method that calculates the new token partitions for a given object
    Args:
        tokens: current number of tokens of the object
        min_number_of_tokens: defined minimum number of tokens
    Returns:
        a partition everytime it's called
    """
    config.number_of_partitions = self._choose_number_of_partitions()
    tkns_per_partition = min_number_of_tokens / config.number_of_partitions

    if len(tokens) < min_number_of_tokens:
        # In this case we have few token and thus we split them
        step_size = ((2 ** 64) - 1) / min_number_of_tokens
        partition = []
        for fraction, to in tokens:
            while fraction < to - step_size:
                partition.append((fraction, fraction + step_size))
                fraction += step_size
                if len(partition) >= tkns_per_partition:
                    yield partition
                    partition = []
                    config.number_of_partitions =
                        self._choose_number_of_partitions()
                    tkns_per_partition = min_number_of_tokens /
                                         config.number_of_partitions
            # Adding the last token
            partition.append((fraction, to))
        if len(partition) > 0:
            yield partition
    else:
        i = 0
        while i < len(tokens):
            splits = max(len(tokens) / config.number_of_partitions, 1)
            yield tokens[i:i + splits]
            i += splits
            config.number_of_partitions = self._choose_number_of_partitions()
```

Anexo 3. Implementación del gestor estático por tamaño de los datos

self._father es el objeto que estamos particionando.

```
def split(self):
    '''
    config.partition_strategy == "TABLE_SIZE"
    Data will be partitioned in table_size // config.optimal_partition_size
    different chunks
    :return: an iterator over partitions
    '''
    table_size = self._get_table_size()
    if table_size == 0:
        config.partition_strategy = "SIMPLE"
    else:
        config.number_of_partitions = ceil(float(table_size) /
                                           float(config.optimal_partition_size))
    return SimplePartitioner(self._father).split()

def _get_table_size(self):
    '''
    :return: the size of the table in kilobytes
    '''
    # system.size_estimates can take time to be refreshed
    output_flush = output_refresh = 1
    for node in config.contact_names + ["localhost"]:
        try:
            # first we need to flush the Cassandra cache, 0 if finished correctly
            output_flush = subprocess.call(
                "nodetool -h {} flush -- {} {}".format(node, self._father._ksp,
                                                       self._father._table),
                shell=True)
            # then we refresh the system.size_estimates table,
            # 0 if finished correctly
            output_refresh = subprocess.call("nodetool -h {}
                             refreshsizeestimates".format(node), shell=True)
        except Exception as ex:
            print("Could not flush data in node {}.".format(node))
            print(ex)
        if output_flush == 0 and output_refresh == 0:
            break
    else:
        print("Could not flush data in any node.")
        return 0

    prepared_get_size = config.session.prepare(
        "SELECT mean_partition_size, partitions_count
         FROM system.size_estimates
         WHERE keyspace_name='{}' and table_name='{}'".format(
            self._father._ksp, self._father._table))
```

```
    res = None
    # some attempts to wait until the cache is flushed and the
    # system.size_estimates table is refreshed
    attempts = 0
    while attempts < 5:
        res = config.session.execute(prepared_get_size)
        if res:
            break
        attempts += 1
        time.sleep(1)

    if not res:
        print("Could not get table size.")
        return 0

    # aggregate info of the table
    total_size_bytes = 0
    for partition_size, partitions_count in res:
        total_size_bytes += partition_size * partitions_count

    total_size_kb = total_size_bytes / 1000
    return total_size_kb
```